



NAVAL POSTGRADUATE SCHOOL

Monterey , California



THESIS

53552

PROTOTYPING VISUAL DATABASE
INTERFACE BY
OBJECT-ORIENTED LANGUAGE

by

Robert J. Schuett
o e o

June 1988

Thesis Advisor:

C. Thomas Wu

Approved for public release; distribution is unlimited

T242335

REPORT DOCUMENTATION PAGE

REPORT SECURITY CLASSIFICATION Unclassified		1b RESTRICTIVE MARKINGS	
SECURITY CLASSIFICATION AUTHORITY		3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited.	
DECLASSIFICATION/DOWNGRADING SCHEDULE			
PERFORMING ORGANIZATION REPORT NUMBER(S)		5 MONITORING ORGANIZATION REPORT NUMBER(S)	
NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL (If applicable) Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000		7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	
NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
ADDRESS (City, State, and ZIP Code)		10 SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO
		TASK NO	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Prototyping Visual Database Interface by Object-Oriented Language			
12. PERSONAL AUTHOR(S) Schuett, Robert J.			
13a. TYPE OF REPORT Master's Thesis	13b. TIME COVERED FROM TO	14. DATE OF REPORT (Year, Month, Day) 1988 June	15. PAGE COUNT 112
16. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
		object-oriented language, graphics interface, database language	
19. ABSTRACT (Continue on reverse if necessary and identify by block number) A graphics user interface called GLAD (Graphics Language for Database) has been proposed as a unified interface method for interaction with a database. The GLAD interface is a graphic object-orient environment which provides a rich intuitive interaction for the user. The interface between GLAD and the user is accomplished through the use of an Object-Oriented Language (OOL). Predefined classes in an OOL provide a robust capacity for quick prototype implementation. The thrust of this thesis is to describe the current status of the GLAD implementation and show how the use of an Object-Oriented Language is an effective tool for implementation of the GLAD interface.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Professor C. Thomas Wu		22b. TELEPHONE (Include Area Code) (408) 646-3391	22c. OFFICE SYMBOL Code 52 Wq

Approved for public release; distribution is unlimited.

**PROTOTYPING VISUAL DATABASE
INTERFACE BY
OBJECT-ORIENTED LANGUAGE**

by

Robert J. Schuett
Major, United States Army
B.S., United States Military Academy, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1988

ABSTRACT

A graphics user interface called GLAD (Graphics **L**anguage for **D**atabase) has been proposed as a unified interface method for interaction with a database. The GLAD interface is a graphic object-orient environment which provides a rich intuitive interaction for the user. The interface between GLAD and the user is accomplished through the use of an Object-oriented language(OOL). Predefined classes in an OOL provide a robust capacity for quick prototype implementation.

The thrust of this thesis is to describe the current status of the GLAD implementation and show how the use of an Object-oriented language is an effective tool for implementation of the GLAD interface.

TABLE OF CONTENTS

I.	INTRODUCTION	1
A.	BACKGROUND	1
B.	OBJECT-ORIENTED LANGUAGES	2
1.	Objects	3
2.	Classes	3
3.	Methods	4
C.	ACTOR LANGUAGE	5
D.	WHY GLAD?	9
1.	Aggregation	9
2.	Classification	11
3.	Generalization	11
4.	Association	13
E.	SUMMARY	13
F.	ORGANIZATION	13
II.	GLAD INTERFACE	15
A.	ENVIRONMENT	15
1.	Representation of Objects	15
2.	Mouse and Its Buttons	16
3.	Window Mechanisms for Interface	17
B.	TOP-LEVEL GLAD WINDOW	18
1.	Starting GLAD	18
2.	Menu Choices	20
3.	Quit	22
C.	DML WINDOW	22
1.	Schema Manipulation	23
a.	Describe	24
b.	ShowConnectn	25
c.	Expand	26
2.	Data Manipulation	29
a.	All at Once	30
b.	One by One	34
c.	LstMemWindow and OneMemWindow Interaction	36
III.	IMPLEMENTATION DETAILS	38
A.	MODULAR CONSTRUCTION	38
B.	BENEFITS OF INHERITANCE	40

C. BENEFITS OF MESSAGE PASSING	41
D. BENEFITS OF POLYMORPHISM	42
E. DISCUSSION OF RELATIONAL DATA MODEL	43
F. CONCLUSIONS	45
APPENDIX A - GLADAPP.CLS FILE	46
APPENDIX B - GLADWIND.CLS FILE	47
APPENDIX C - DBDIALOG.CLS FILE	49
APPENDIX D - DMWINDOW.CLS FILE	53
APPENDIX E - DSCRWIN.CLS FILE	62
APPENDIX F - LSTMEMWI.CLS FILE	66
APPENDIX G - ONEMEMWI.CLS FILE	76
APPENDIX H - NESTDMWI.CLS FILE	83
APPENDIX I - CONNOBJW.CLS FILE	84
APPENDIX J - DBSCHEMA.CLS FILE	88
APPENDIX K - GLADOBJ.CLS FILE	90
APPENDIX L - COLORTAB.CLS FILE	93
APPENDIX M - GLAD RESOURCE SCRIPT FILE	94
APPENDIX N - CONSTANTS AND GLOBAL VARIABLES	97
APPENDIX O - UNIVERSITY DATABASE FILES	99
1. UNIVERS.SCH FILE	99
2. EMPLOYEE.DAT FILE	101
3. DEPT.DAT FILE	101
LIST OF REFERENCES	102
INITIAL DISTRIBUTION LIST	103

LIST OF FIGURES

Figure 1.1	GLAD Class Tree	6
Figure 1.2	University database	10
Figure 1.3	Description Window	11
Figure 1.4	Generalized Objects	12
Figure 2.1	Hierarchical Structure of Commands	19
Figure 2.2	Start Window of GLAD	20
Figure 2.3	Available GLAD Databases	21
Figure 2.4	DML Window	23
Figure 2.5	Multiple Description Windows	25
Figure 2.6	Showing Relationships	27
Figure 2.7	Multiple Nested DML Windows	28
Figure 2.8	LstMemWindow and OneMemWindow Windows	32

ACKNOWLEDGEMENTS

I would like to thank Dr. C. Thomas Wu for giving me direction, assistance, and confidence in my endeavor to write this thesis. His understanding of the object oriented language ACTOR and message passing with Microsoft Windows was shared and is greatly appreciated.

I also thank my beautiful wife, Doreen, for her love, encouragement, and support throughout my career and especially the months I studied for my masters.

Last, but not least, is my daughter, Ryan Elizabeth, for her smile and laugh each night I put her to bed and went back to my studies.

I. INTRODUCTION

A. BACKGROUND

The impact of computers on society is still expanding. The memory capacities of computers, and especially personal computers (PC), have increased the capabilities and productivity of companies and individuals. Word processing, communication networks, electronic mail and database management systems (DBMS) are no longer limited to a select population of computer experts.

A more diverse range of people want to use computers and use them easily. Users of DBMS can interact with databases in three different ways [Ref. 1]:

- *Data definition interaction* is the creation of a database¹ via data definition language.
- *Data manipulation interaction* is the accessing (ie. retrieval and update of information) of database via a data manipulation language.
- *Program development interaction* is the development of application programs via an embedded host language.

Developing an effective and usable, yet easily learned and used, DBMS is our concern.

As more non-technical people use computers, the more there is a demand to use the power of a computer without having to understand what makes it work. One area of research and development has been in some fifth generation languages that will provide a natural extension of the human mind to do problem solving programming. The desire is to provide a programming language that has a close correspondence between physical and logical entities of the program model and the real world problem. This evolution is

¹ We mean the definition of the database schema and not the actual loading of the database.

object-oriented languages (OOL). Object-oriented languages have been developed in a graphics oriented environment that can provide rich interaction for the user.

A new graphics user interface, **Graphic Language for Databases (GLAD)** has been proposed and discussed [Refs. 1,2]. **GLAD** will be able to accommodate both sophisticated and naive users. A database definition language (**DDL**) has been proposed [Ref. 3]. The thrust of this thesis is to show that the object-oriented language is an effective tool for implementation. The implementation and programming work supporting this thesis is done in an object-oriented language known as ACTOR.²

The remainder of Chapter I introduces some of the important concepts of object-oriented languages and specific details of Actor. Then we finish with a discussion of three of the abstraction concepts supported by **GLAD**.

B. OBJECT-ORIENTED LANGUAGES

GLAD is unique in that our approach is to use object-oriented programming. Object-oriented languages are characterized by three criteria [Ref. 4:p. 1.2.1]:

- encapsulation of data and instruction into units of functionality called objects
- dynamic binding (at run time) of messages and the corresponding methods, allowing for more flexible code
- inheritance of methods through hierarchy scheme

Users of OOL must learn new words and concepts when dealing with *objects* as an encapsulated entity that contains both the data and instructions. Instructions are referred to as *methods*[Ref. 5:p. 31]. Objects are self-contained and self-governed.

This feature of OOL supports the Principle of Information Hiding [Ref. 6]. And as modern programmers know, information hiding reduces the interdependence of different

² ACTOR and ACTOR LOGO are trademarks of The Whitewater Group, Inc.

parts of application software. The implementor of software packages does not know how the user will use it, and the user does not have complete details on how the package is implemented [Ref. 4]. Programmers also know that this principle helps in code reliability, extensibility and maintainability.

We have mentioned unique words and concepts about OOL. We now discuss the different characteristics of *objects*, *classes* and *methods* in OOL.

1. Objects

Object-oriented programming allows us to represent problems and solutions in a new way. The creation and management of *objects* is the basis of object-oriented programming. An object can be designed to closely resemble real-life objects. Our objects have attributes and respond to messages. In OOL *everything* is an object. Even data structures like integers, arrays, characters and rectangles are all objects. In OOL we do not have separate data and separate commands as in procedural languages. The instructions are part of the object and are designed according to the particular internal format of the object. Programmers can think in terms of the attributes of objects and the instructions which the objects will respond to. [Ref. 4]

2. Classes

Now that we have discussed object-oriented concepts and have presented the fact that objects control the data of a program and the operations on that data, we need to discuss different types of objects. We say that a type is a *class* of objects. There can be many different classes, but each object can belong to one and only one class. The relationship between different types of objects is an important characteristic of OOL's. The class of an object defines exactly what property that object will possess and how the object will respond to instructions, known as *messages*. We will present you with an object called employee in our GLAD interface. Our particular employee, Joe Doe Jr., is an *instance* of our abstract employee object. The employee object is an instance of our

abstract class GLADOBJ.CLS. Think of our GLADOBJ.CLS as an object template for a **GladObj** object. Every object of a given class has the same data format and responds to the same instructions. Each instance of a **GLAD** object has information of its own, such as a **GLAD** object's name, point, color, nesting, attributes, etc. In object-oriented terms, this information is stored in *instance variables* and is referred to in this way because every instance of a class has its own copy of them. The data portion in each object, or instance of a class, is private and owned by the instance. The instruction portion of an object is owned by the instance class. [Refs. 4, 5]

3. Methods

We have mentioned that everything in OOL is an object, every object is an instance of a class and instances of some classes carry some of their data in their instance variables. We also mentioned something about instructions being part of an object and shared by class instances. We need to mention how objects interact to do the things we want. Since object-oriented languages have their data and code that works on the data bundled together, we have to send a *message* to an object to have the code executed. So to get an object to do something, we send a message to the object. In OOL we do not refer to these instructions as functions or procedures. In object-oriented terms these instructions are called *methods*. A method is tailored so it can only act on a particular object. A *class* method is the only one authorized to handle an object. We may have many methods that perform the same operation, such as *start*, but the operations which the methods perform are data-specific or object-specific. OOL methods and classes prevent errors that occur when an operation on the wrong type of data is performed. [Ref. 4]

It is important to understand the distinction between messages and methods. These two terms are intimately related in object-oriented languages, but are different

concepts. A *message* represents a request to perform an operation, or instruction. A *method* represents the implementation of an instruction for a particular class. Using the run time power of inheritance, implementation can be done by the class's descendant classes. [Ref. 4]

C. ACTOR LANGUAGE

We are using the new object-oriented language called ACTOR which runs under Microsoft Windows(MS-Windows).³ Actor can be used on any computer that can run MS-Windows and is equipped with a hard disk, 640K RAM, a graphics display with adapter and a mouse [Ref. 5:p. 1]. The development and utilization of this portion of **GLAD** using Actor was conducted on a Zenith 248 computer with a 20Meg hard disk, MS-DOS Version 3.20 and Microsoft Windows Versions 1.03 and 2.03.

For a program to use objects of a class, the program must contain the class definition. This is referred as the class "must be loaded into the system." [Ref. 4:p. 1.1.2] Actor comes with over 100 predefined classes [Ref. 4:p. 1.4.2] of which about 80 classes are loaded in the system [Ref. 5:p. 118-119]. Our program uses many of these classes, such as OBJECT, COLLECTION, RECT and WINDOW Classes. We have defined and loaded thirteen new classes, see Appendixes A-L, to execute our program. All the classes we have written are descendents of an Actor class,⁴ as shown in Figure 1.1. This allows our classes to share the operations of any ancestor class and we only have to change some of the behavior. In **GLAD** we have written descendant classes of the WINDOW Class hierarchy. The Actor Window Classes define the default behavior for

³ MS and Microsoft are trademarks of Microsoft Corporation.

⁴The actual Actor class hierarchy is much richer than shown in Figure 1.1. We have shown the part of the Actor hierarchy that is in the ancestry of our GLAD classes.

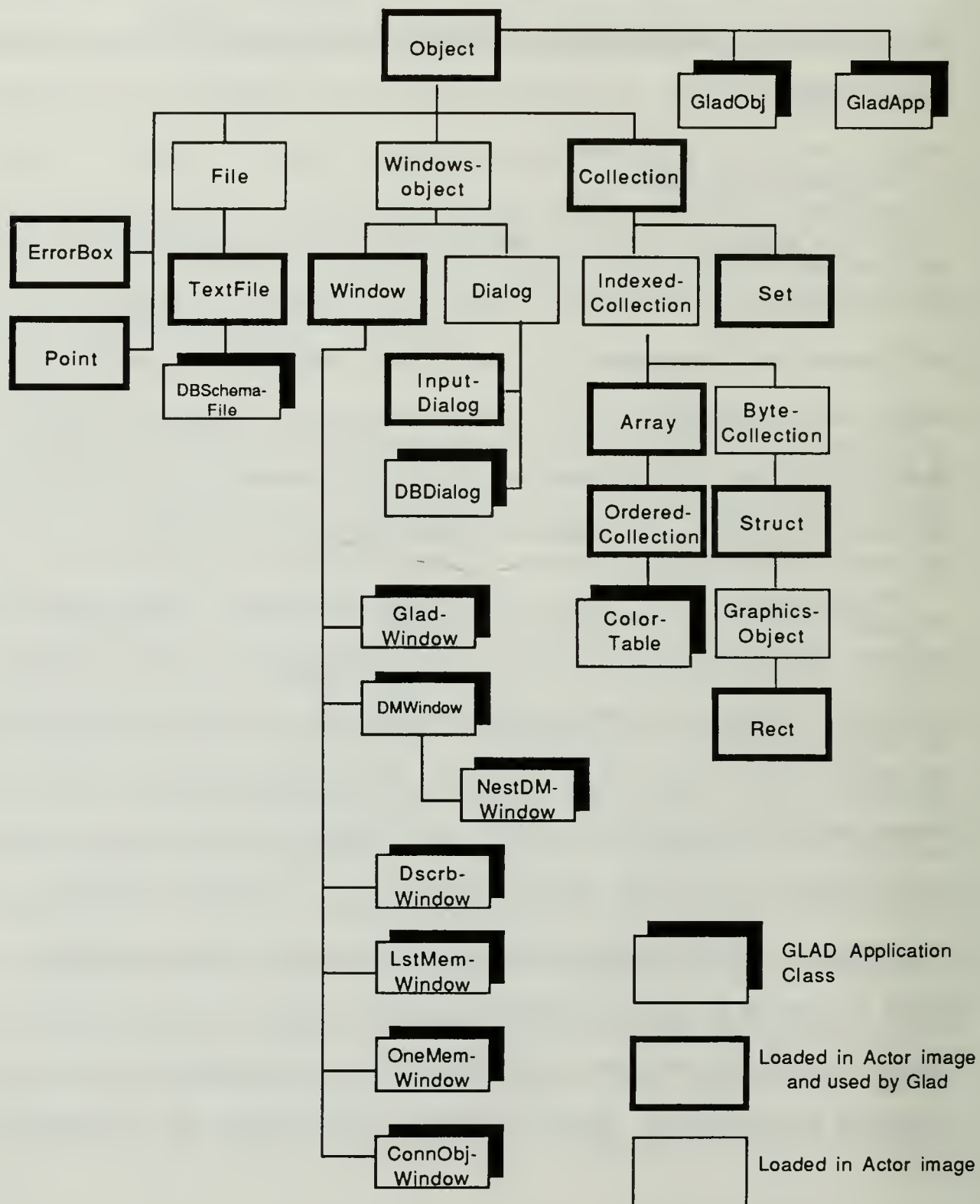


Figure 1.1 GLAD Class Tree

all windows, such as create and display graphics. Our new classes refine the functionality to suit our program.

We have mentioned that object-oriented programs consist of objects sending messages to each other. To write our program in Actor was just a matter of designing the layout of the objects and writing the method that the object will execute [Ref. 5:p. 49]. The last part of this section will explain some basic details of Actor method and message syntax.

When a program is running, each message must be matched up with a *method*. Every Actor method has the general format shown here [Ref. 5:p. 49]:

```
/* Method comment */
Def methodName(self,arg1,arg2,... | loc1,loc2,...)
{ statement1; /* comment */
  statement2;
  .
  .
  .
  statementN; /* comment */
}
```

The **/* Method comment */** line is an optional, but recommended, piece of text that explains to anyone reading the code what the method is suppose to do. Actor's way of delimiting comments between **/*** and ***/** is just like those in C language. Actor allows the programmer to be very free with comments. [Ref. 5:p. 49-50]

The second line is the method header. The key word **Def** prepares Actor to compile a method. By convention, not compiler enforcement, the name of the method starts with lower case letters and every new word after the first is capitalized. The word **self** refers to the receiver of the message. Since the receiver is not known when the method is written, **self** represents the object that will be sent the message. **Self** refers to the receiver object when it is used as a variable in the method. **Arg#** represents, if any, the arguments

or parameters sent with the receiver. The optional `|` is only needed to separate any local variables for the method from the arguments which are passed. A maximum of fifteen(15) arguments and local variables per method is allowed. The left curly, `{`, and right curly, `}`, brackets signify the beginning and end of the method code. All statements, except the last one, are required to have the semicolon(`;`) at the end. [Ref. 5:p. 50]

Every method returns a value. All Actor methods return *self* as the value unless otherwise overridden. To return an explicit value from a method, Actor uses the caret character, `^`. Whether a method has one or several `^` characters, the first `^` that Actor encounters causes it to immediately exit the method and return whatever is following the `^` character. [Ref. 5:p. 50-51]

We invoke a method associated with an object by sending a message to the object. A message has three elements. The syntax for a typical message,

```
messageName(receiver,arg1,arg2,...);
```

consists of the message name followed by the receiver object name and any argument object names [Ref. 4:p. 1.1.6]. The name of the message is identical to the name of the method being invoked. There is no requirement for the syntax to have the receiver object's class. The power of inheritance in Actor allows us to specify only the object's name. The ability to send the same message to different objects and have the correct method executed is referred to as *polymorphism*. This powerful concept closely parallels the way we think and represents the enormous flexibility in Actor programming. To use the concept to its potential, code is written that specifies only general instructions and delegate to the object involved the handling of implementation details. This way we need only substitute one object for another to reuse the code. We believe this aids in easier maintenance and reuse of code. [Refs. 4,5]

Another way of representing a message is by using infix format. This is used for common arithmetic and logical operations, such as:

```
aClientPt.x * 1024/wd  
attr[NAME] + stringOf('... ')  
DT_VCENTER bitOr SINGLELINE
```

In Actor, the symbols like *****, **+** and **bitOr** are messages to an object. This type of message is specially handled to work in the infix format. In the example above, **1024** is the object receiving the ***** message. [Ref. 5:p. 147-148]

D. WHY GLAD?

GLAD will provide us with the means to elegantly display real world abstract concepts. We have utilized a bit-mapped, high-resolution graphics display terminal. The screen consist of two types of **GLAD** windows: *schema* and *operation window*. Both windows have the same basic structure and can be opened, closed, scaled and moved by the user. Figure 1.2 is our sample **GLAD** schema that will be our example to show how each abstraction concept is usually represented and how we illustrate the **GLAD** describe feature of an operation window in our *data manipulation interaction*. As stated in [Ref. 1], we want to provide a visual representation for the four most widely used abstraction concepts: aggregation, classification, generalization and association.⁵ In this section we discuss and show how the first three concepts are represented in a **GLAD** diagram.

1. Aggregation

An object⁶ is an *aggregation* of (sub)objects [Ref. 2:p. 6]. Our example of an

⁵ It is not our intention to provide a visual aid in explaining the meaning of these four abstractions. We are devising visual representation which are suitable for direct manipulation.

⁶ The term object is not defined. We continue to view an object as a "thing" (tangible or intangible) that exists.

employee object is an aggregation of name, age, pay, address and worksfor (sub)objects. An aggregate object is a rectangle in a **GLAD** diagram, as shown in Figure 1.2. The (sub)objects of an aggregate object can be seen with our **Describe** window, as shown in Figure 1.3. This operation window for description is displayed in the bottom right quadrant of our schema window. Notice that our window capabilities allow us to highlight our equipment object in the schema window and the border shadow of our description operation window with the same color.⁷ Also, as presented in [Ref. 1] the *non-atomic* aggregate **dept** object in the operation window is color highlighted and the

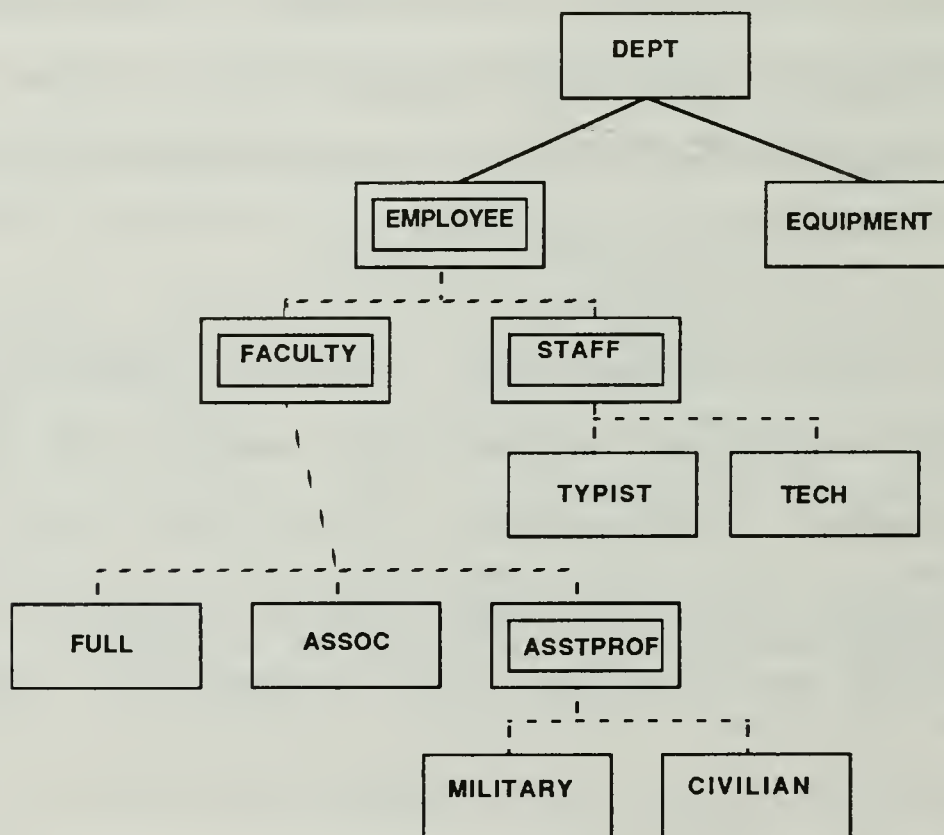


Figure 1.2 University database

⁷ Different shade patterns will be used in the mono-graphics monitor.



Figure 1.3 Description Window

corresponding **dept** object in the schema window now has the same color. Further details of this window will be discussed in Section C of Chapter II.

2. Classification

Classification simply means that each data item in a database is information about the object it belongs to. Our data item [Joe Doe Jr, 14, 23,000, 8900 Coker Rd..., Forestry] is an information about employee, and is *classified* as an employee object. Each data item of an object is called an *instance* of that object. [Ref. 1]

3. Generalization

Faculty and staff are individual objects that we have grouped together to form a *generalized* object called employee. We characterize the generalization abstraction as an

IS-A relationship. In our university database, faculty IS-A employee and staff IS-A employee.

A nested rectangle is the representation of a generalized object in a *GLAD* diagram, as shown in Figure 1.4a. The user need only issue an expand command and view the specialized objects, as shown in Figure 1.4b. As depicted in Figure 1.4b, the specialized objects can also be generalized objects of further specialized objects. Our **faculty** object is a generalized object of full, associate and assistant professors. The **assistant professor** object is its self a generalized object of additional individual objects. [Ref. 1]

As mentioned in the previous section, an association between two objects is representative by a solid line. If one or both of the associated objects are specialized objects, we use a dotted line, as shown in Figure 1.2. Graphical representation of different types of objects and in a more intricate fashion, as discussed by Wu [Ref. 1], are still being developed to satisfy the overall design of *GLAD*.

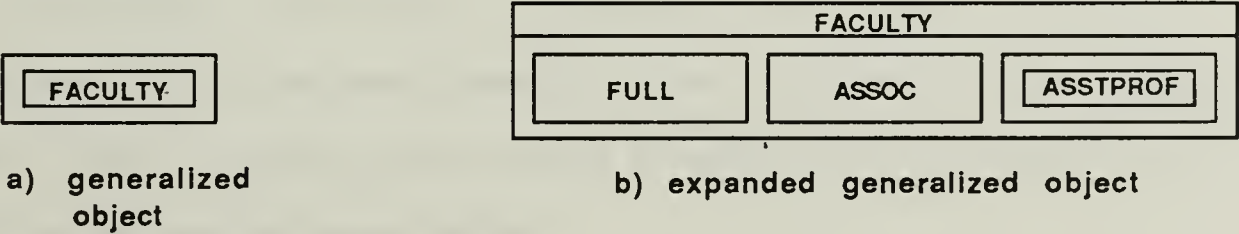


Figure 1.4 Generalized Objects

4. Association

The fourth abstraction, *association*, is not discussed in this paper. The reader should refer to [Ref. 1] on how association will be handled within **GLAD** as we continue to develop this graphics user interface.

E. SUMMARY

As an object-oriented language, Actor is a significant departure from procedural programming. Actor has two cardinal rules of programming: [Ref. 5:p. 35]

- EVERYTHING in Actor is an object. Numbers, characters, strings, applications, windows, methods, and so on are all objects.
- Every action that occurs in Actor (except MS-Windows or MS-DOS) is the result of sending a message to an object, which responds to it by executing a method. There are no other exceptions beside those mentioned above.

Polymorphism in object-oriented programming provides us the ability to reuse code.

Objects are arranged hierarchically in classes and any descendant classes inherit the behavior of their ancestors. We have also continued Actor's convention that method and instance variable names start with lower case characters and that global, or system variables, are capitalized.

F. ORGANIZATION

The remainder of the thesis is presented in two chapters. Chapter II is an introduction to the current states available for *data manipulation interaction* within the **GLAD** interface. A sample **GLAD** session is used to show the current state of the application and user interface capabilities.

Chapter III concludes this paper with implementation details and benefits gained in the **GLAD** interface by use of an object-oriented language. The chapter also contains conclusions on the use of object-oriented programming in further developments of our **GLAD** interface. The Actor code developed for this application by the author and

Professor C. T. Wu is contained in Appendices A through N. Appendix O contains the files used to create the sample session and its instances of each **GLAD** object. Each appendix contains information for proper recognition of authorship of code included with this thesis.

II. GLAD INTERFACE

We will now illustrate the current states and features of **GLAD** by going through a sample session in the sections of this chapter. We will show all active features of **GLAD** that the user can presently manipulate on the screen. We will stop at the point where the user would begin to change the database and/or formulate his subqueries within the database. Dummy database files were used to develop and illustrate the current abilities of **GLAD**'s visual representation on the screen of a user's retrieval of a database (see Appendix O).

A. ENVIRONMENT

1. Representation of Objects

As presented in Chapter I, our **GLAD** object, an aggregated object, is represented as a rectangle on the screen of the monitor in a **GLAD** diagram. Individual objects are seen as a single rectangle with the name of that object printed and centered within the boundaries of the rectangle. If the object is the grouping of specialized objects, we refer to it as a *generalized* object and represent this abstraction as a nested rectangle, a rectangle within a rectangle. When these rectangles are initially presented on the screen, they are drawn with standard width border lines and 'filled' with white color. If the **GLAD** object is selected by the user or referenced by another object during the application interface, its color is changed to signify some interaction has occurred. A color is used to reference one **GLAD** object and any association with that object during data manipulation.

When a **GLAD** object is selected by the user, the outer border line of the rectangle is made **bold** to visualize to the user his *most recently selected* object within a

given window. When the user selects another object or de-selects the most recently selected object, the bold line of the rectangle is returned to standard width. Selection and de-selection is discussed in the next section of this chapter.

2. Mouse and Its Buttons

Since the mouse interface is an important part of all MS-Windows applications, users of **GLAD** need a mouse. Our application presently uses two buttons of the mouse, the left button and the right button. The flexibility of **Glad** and the use of our object-oriented language will allow us to expand the use of the third mouse button (middle) in follow-on developments. Any user familiar with the use of a mouse in conjunction with MS-Windows will transition easily in the manipulation of **GLAD** windows and objects.

The user clicks⁸ the left button when he wants to select something on the screen. He uses the left button to select a menu choice, acknowledge button and highlight an object or an *instance* of the object whose data items are displayed on the screen. If the user wants to reposition an object to another position on the screen, he need only hold down the left button while the cursor is on the object and *drag* the object across the screen. If the user uses the left button inappropriately, he will be helped with an **Error Message**.

The present state of the right button is to de-select the *most recently selected* highlighted object or highlighted line within a List window. If the user forgets and uses the right button inappropriately, he will either get no response or be prompted by one of several **Error** boxes with an appropriate message to help the user.

⁸ Unless otherwise noted in this chapter, *click the mouse*, *click on* or *select* implies the user is pressing and releasing the left button of the mouse.

Having the buttons perform the same function in all windows supports two of our characteristics of a good user interface for data manipulation [Ref. 2:p. 5]:

- It must be easy to learn. Naive and uninitiated users should be able to master the interaction method quickly and start accessing the database with a short learning period.
- It must be easy to use. It must be easy to use so users rarely make erroneous queries and

3. Window Mechanisms for Interface

Since **GLAD** interacts with MS-Windows and its mouse applications, we have used several common features with our screen for the user to interact with **GLAD**. We have incorporated the use of push buttons,⁹ dialog boxes, error boxes and menu choices within the windows. Each of these mechanisms for interface with the user mimics what the user sees in standard MS-Windows.

GLAD presently uses push and default push buttons. The user is presented buttons when he encounters most of the dialog boxes presented to him throughout the application. The **start** push button is the first one the user sees when he enters the **GLAD** application.

Users of our **GLAD** application are presently presented three different dialog boxes. These are the two dialogs that are presented from our top-level **GLAD** window for the *Open* and *Remove* menu options and one for our *GoTo* menu option in the **One Member** window, see Figure 2.1. Examples of these are presented in our sample session later in this chapter. We also use one of the MS-Windows simpler dialog boxes throughout the **GLAD** application. Any time the user tries to execute an inappropriate

⁹A push button is a small area on the screen that the user can click with the mouse to invoke some response in the application. A *default* push button looks like a normal push button except its edges are displayed on the screen as thicker lines to indicate that it is the default choice. In our **GLAD** application default means the user can click it with the mouse or he can press the Enter/Return key on the keyboard to activate that button.

menu option or input invalidate data (e.g., position number for data element), an `ErrorBox` object is presented with a message to assist the user.

Figure 2.1 shows the hierarchical structure of **GLAD** commands. All commands presently available to **GLAD** are shown here. Some of the commands will not be discussed. The user of the **GLAD** interface is presented three different ways to view the available commands. The user selects his desired command from those available in the menu bar at the top of a window created on the screen. Each window has its own menu options.¹⁰ He is also presented a drop-down menu from certain menu bar options, see the third level menu in Figure 2.1. This allows us to group similar commands under one common reference, which assists in maintaining a more user-friendly menu bar. The third way is using the buttons presented in the dialog boxes.

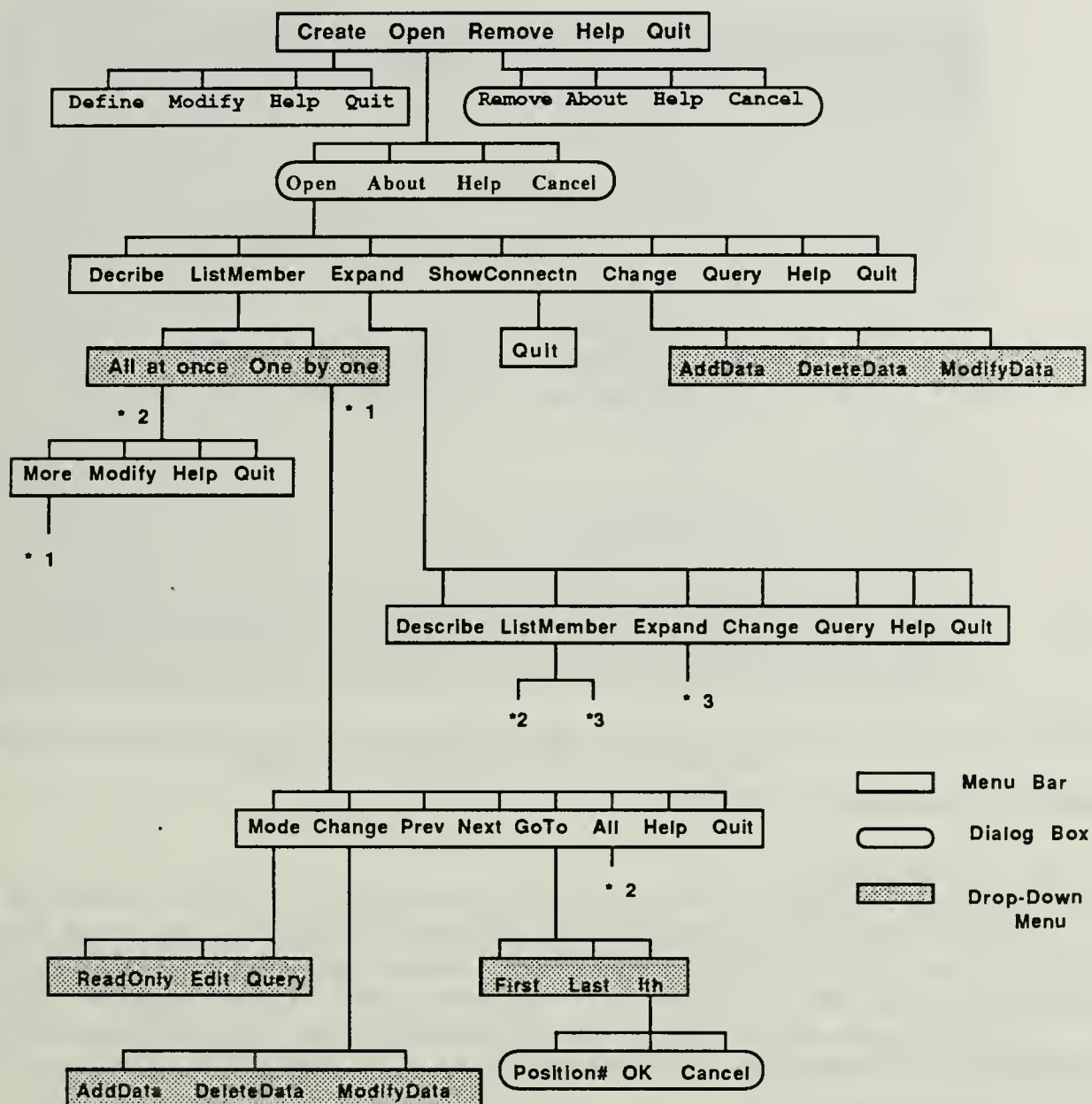
This thesis will deal mainly with the commands that fall under the **OPEN** command within the top level commands. Detail discussion of each level of commands is presented in the following sections of this chapter and clarification is made through the use of a sample session with all active commands.

B. TOP-LEVEL GLAD WINDOW

1. Starting GLAD

Figure 2.2 shows us the **GLAD** window the user first sees when the program is executed. This is the top level window of **GLAD** and is so annotated in the window title at the top of the screen. The **Glad** window and **Dialog** box with the `ABOUT_GLAD` text are shown in the figure. The **GLAD** window is disabled until the `ABOUT_GLAD` dialog is closed. The user must click on the dialog box's **START** button to allow the **GLAD**

¹⁰Menu options for each window are stored in file `Glad.rc`, Appendix M.



- *1 Initiates a new "One by One" window or brings to the top if previously initiated
- *2 Initiates a new "All at Once" window or bring to the top if previously initiated
- *3 Expand generalized object, continuous as long as a generalized object is present at sub-level

Figure 2.1 Hierarchical Structure of Commands

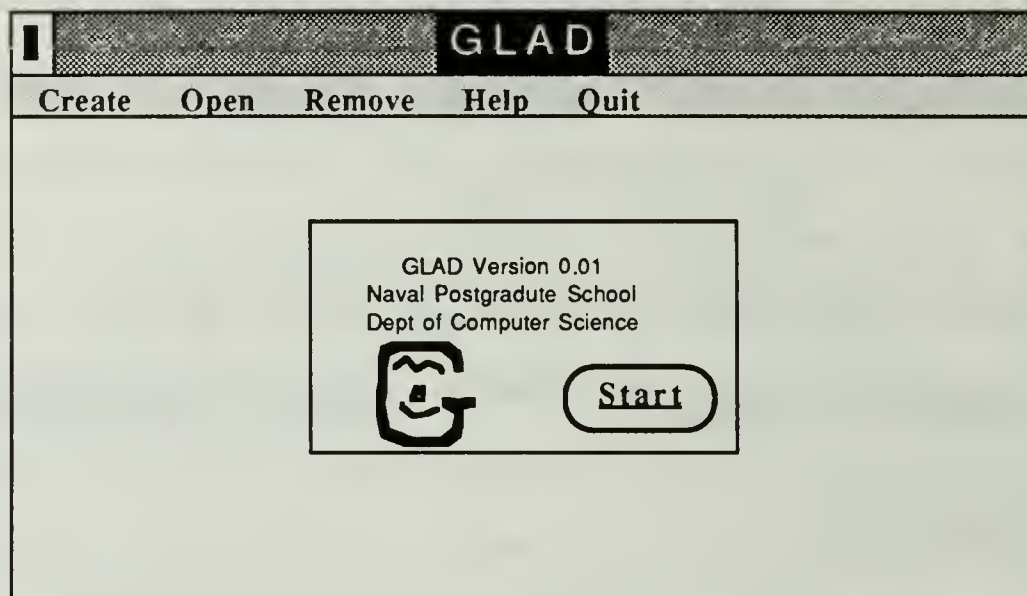


Figure 2.2 Start Window of GLAD

window to be active. Once he has done this, the dialog box is closed and we are ready to begin executing *GLAD* commands.

2. Menu Choices

The menu options available at the top-level of *GLAD* can be seen in Figure 2.1 and in the menu bar of Figure 2.2. All these menu choices apply to the user's manipulation of the database. The user may create a new database by clicking on *Create*. This will present him with a new *GLAD* DDL window that presently is not active except for the user being able to create it and close it with the *Quit* menu option. Closure of the *GLAD* DDL window returns the user to the *GLAD* window.

Both the *Open* and *Remove* options cause a dialog box to be presented to the user with several buttons within the box. Figure 2.3 shows the dialog box presented when the user selects the *GLAD* window *Open* menu option. This option is discussed fully in the next section of the chapter. The *Remove* option presents a dialog box that is

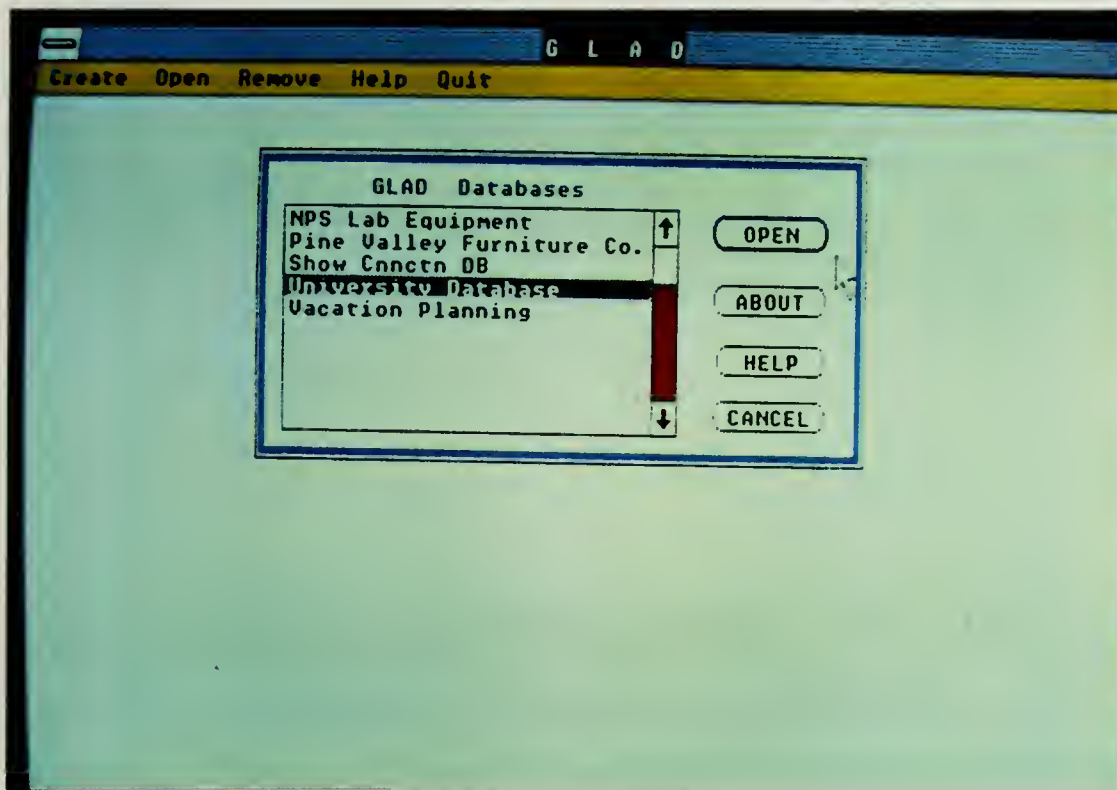


Figure 2.3 Available GLAD Databases

exactly the same as Figure 2.3 except where we see an *OPEN* default push button in the figure is a *REMOVE* default push button. The common *ABOUT*, *HELP* and *CANCEL* buttons within the dialog box are discussed in the section on the *Open* menu option.

The *REMOVE* button within the *Remove* menu option dialog box allows the user to delete any database file from his **GLAD** database files. The user first selects the desired file name within the dialog box. He then can click on the *REMOVE* default push button or press the return/enter key on the keyboard. The application will present the user with another dialog requesting confirmation or cancellation. If the user confirms the command, the highlighted file name is removed from his **GLAD** database files. After the confirm or cancel option, the dialog is closed and the **GLAD** window is again active.

The *Help* option of the **GLAD** window is presently a default dialog box that will be developed to provide assistance to the user. At present, the application responds to this option with a dialog box, a short message and an *OK* button.

3. Quit

The last option available in the **GLAD** window is the *Quit*. This option closes all active parts of the application and returns the user to the host system. At present, the user is presented a dialog the appears to confirm the removal of a database if requested during the session.

C. DML WINDOW

As previously stated, the *Open* option of the **GLAD** window presents the user with a dialog box, as shown in Figure 2.3. The buttons are self-explanatory. If the user wants a brief description of a given database file, he need only click on the appropriate file name and then click on the **ABOUT** button. An **ABOUT** dialog box will appear for the user and give a brief description of that file. For our sample, we have selected the *University Database*, which is highlighted within our dialog box. If the user should select **OPEN** or **ABOUT** without selecting a database file, he would receive assistance via an Error Box stating *Database was not selected properly*.

The user enters the **OPEN** mode by selecting the **OPEN** command from our second level menu within the dialog box. This gives the user our **GLAD Data Manipulation Language (DML)** window and our third level menu.

The DML window is created with the GladDmlMenu menu, see Appendix M, and "GLAD DML" in its title box. The new **DML** window, see Figure 2.4, is imposed over our **GLAD** window. The user need only look at the top of the window and he always knows what level of *data manipulate interaction* he is at. Created along with the **DML** window are the three objects. As shown in Figure 2.4, they are **dept**, **employee** and

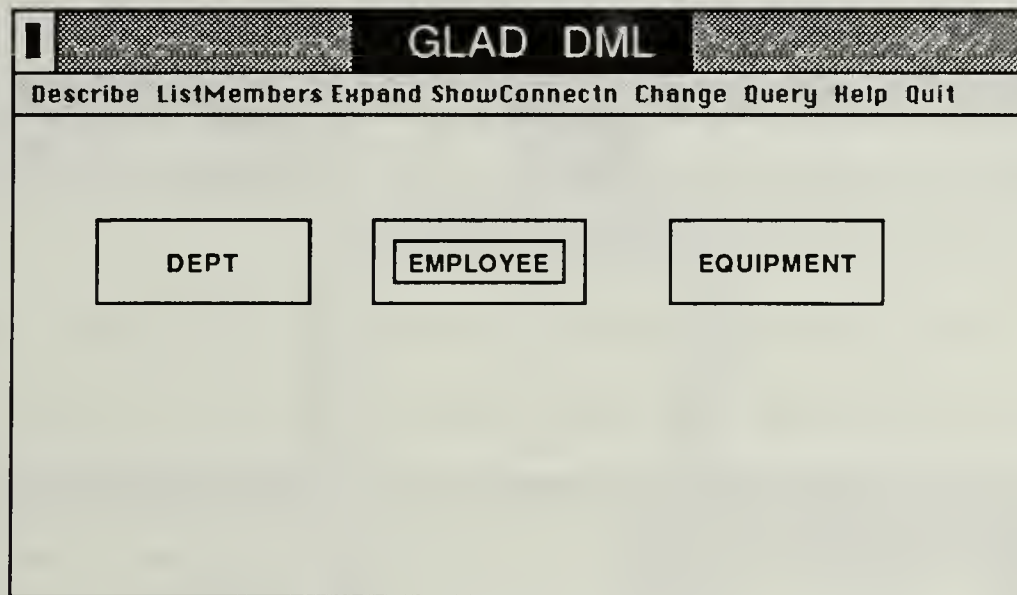


Figure 2.4 DML Window

equipment. The creation of the DML window gives the user manipulation of both the database schema and data.

To initiate menu options *Describe*, *ListMembers* and *Expand*, the user must select one of the objects within the window. Failure to do so will cause the **Glad** interface to initiate a message for an Error Box which states *No object selected*. The user has now begun to view the schema of the University Database (content of schema file *Univers.sch* is listed in Appendix O). Our sample session is based on the data in this file.

1. Schema Manipulation

The user may see and understand more of the schema by selecting the menu options *Describe*, *Expand* and/or *ShowConnectn*. The *Describe* option is for the user to visualize the aggregation of (sub)objects of the selected object. The *Expand* option is for viewing the specialized objects of a generalized object. The *ShowConnectn* option is to show any relationships between objects within the schema.

a. Describe

If the user now clicks on the **employee** object, it changes from white to some color (in this case green). The user can now click on the menu option *Describe* and see the aggregated (sub)objects of our **employee** object, as shown in Figure 1.2. Note again the bold lines of the rectangle representing the **employee** object. This bolder line shows the user his *most recently selected* object.

The user is not limited to just one instance of our **Describe** window. The user may make any number of successive object selections followed by the *Describe* menu option. Subsequent creations of **Describe** windows will be stacked in the bottom right corner of the screen with a slight up and left offset from the previous **Describe** window. The maximum number of **Describe** windows he can have in existence corresponds to the number of objects present in the schema window. Any successive calls of an existing **Describe** window will only have that window brought to the top view of all other windows present on the screen. If the user successively selects *Describe* menu option for each of our three objects and *drags* each object to where he can see each of them, he sees Figure 2.5. As mentioned earlier in the paper, each border shadow of a **Describe** window corresponds to the highlighted color of its respective object. Also, each *non-atomic* aggregate object in a **Describe** window is color highlighted with the same color of its corresponding object in the schema window. This is an effective referencing capability developed between the schema window and **Describe** window. This is a departure from the relational model of databases. The discussion on the relational data model is continued in Section E Chapter III.

Each of the **Describe** windows can be destroyed or removed from the system, by two methods. One is by double clicking on the upper left corner box of each respective window. If the **Describe** window's respective object in the **DML** window is

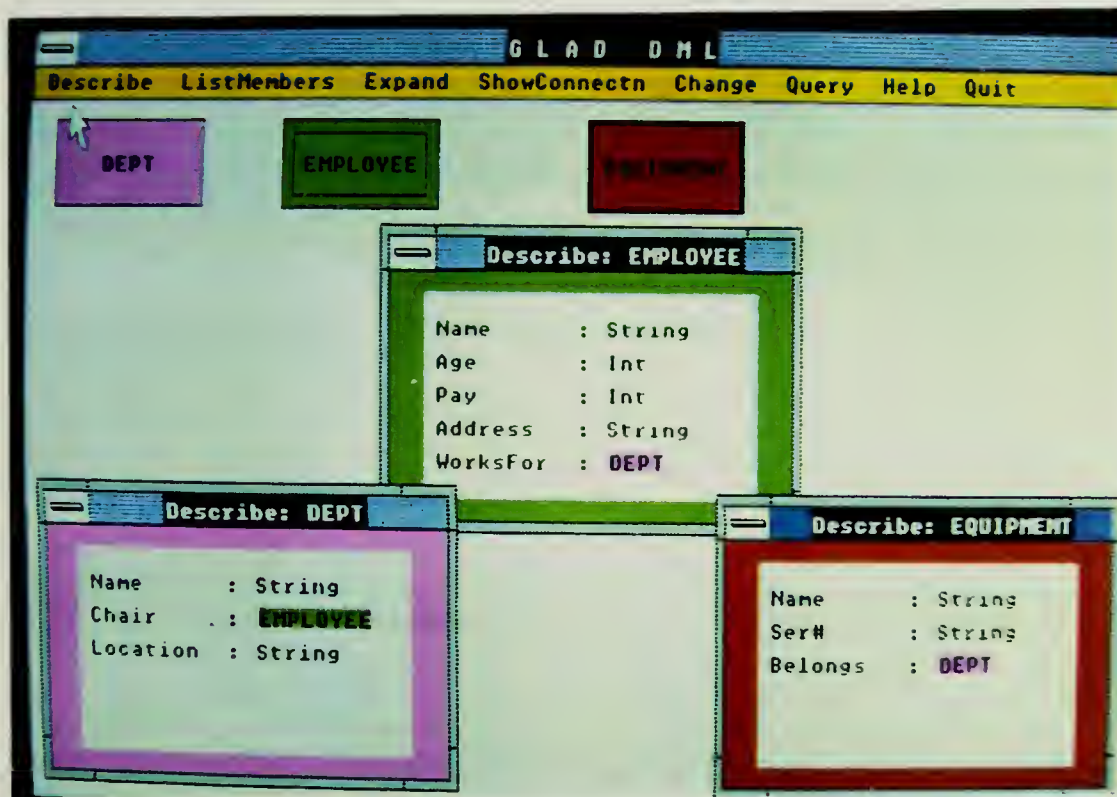


Figure 2.5 Multiple Description Windows

not referenced by another instance of an object or is not the *most recently selected* object, the object is changed back to white. The second method available to the user is de-select the object in the schema window with the right mouse button. The user must ensure that the object he de-selects is the *most recent selected* object before clicking the right mouse button. Failure to do this will initiate an Error Box that tells him *Right button clicked object is not the selected (bold-line) object*. In both cases the user watches the **Describe** window disappear and the object refilled white if not otherwise referenced.

b. ShowConnectn

If the user studies the **Describe** windows in Figure 2.5, he can visualize three relationships between the objects in the schema window. **Employee** object has a

relationship with **dept** object, **equipment** object has a relationship with **dept** object and **dept** object has a relationship with **employee** object. This is easily visible to the user because each aggregate (sub)object's type that is user defined and its corresponding object in the schema window are highlighted with same the color to show a relationship. If the user wants to reinforce his understanding of these relationships or initially wants to see the relationships, he need only click on menu option *ShowConnectn*. If the user selects this menu option while the color of each object is white, he sees the **Show Connection** window presented in the upper photo of Figure 2.6. If the user wants to reinforce his understanding after displaying the three **Describe** windows, he selects *ShowConnectn* and sees the **Show Connection** window presented in the lower photo of Figure 2.6. In both figures the user sees the relationship represented with a solid line between objects. When the user is finished viewing the visual representation, he need only click on the *Quit* menu option and the window disappears from the screen.

c. Expand

As previously depicted in Figure 1.1 and represented in Figure 2.4, **employee** object, a nested rectangle, is a generalized object composed of several levels of specialized objects. We stated that there is a IS-A relationship in our University Database. The IS-A (generalization) hierarchy is easily visualized by this menu option. Our **GLAD** interface user can expand this relationship by clicking on the *Expand* menu option. If the user does not select an object to expand or selects a non-generalized object, he is assisted with an appropriate Error Box stating *No object is selected* or *Selected object is not nested*.

Ensuring the **employee** object is the most recently selected object, the user can click on the *Expand* menu option to see its specialized objects. From Figure 1.1, the user knows that the **employee** object will expand out to two associated objects, **faculty** and **staff**. A nested schema window is created with a title box of "SubClass of: <selected

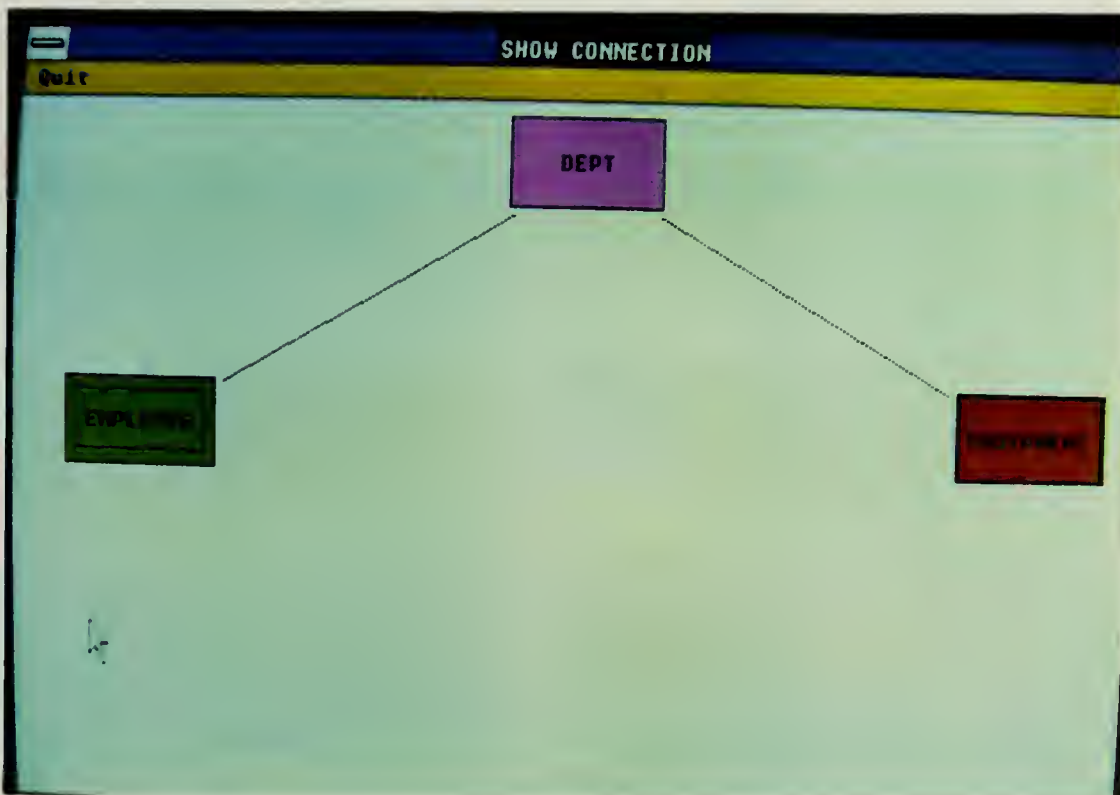
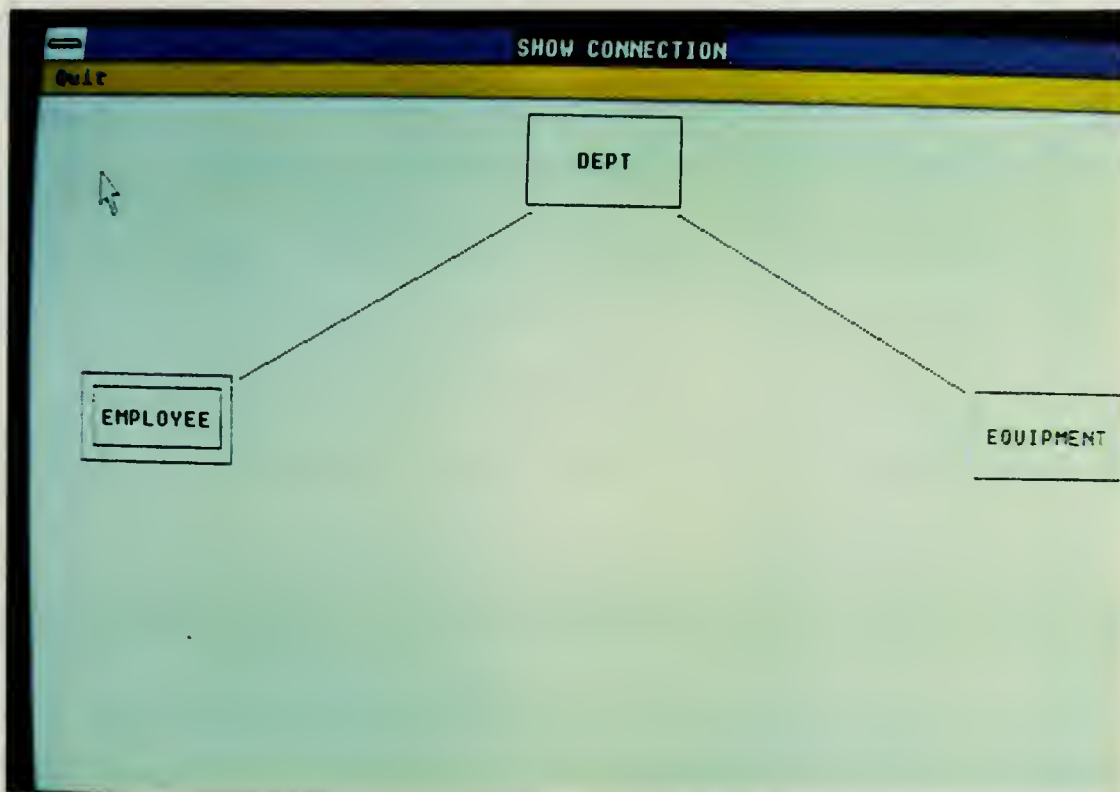


Figure 2.6 Showing Relationships

object's name>", as shown in Figure 2.7. This new window has all the menu options that the DML window has except for *ShowConnectn*. For now, the user will not have this option below the DML window.

The user has a choice of closing all or some of his windows by clicking *Quit* for each individual window he does not desire — in opposite order of creation, or clicking *Quit* on the parent window of those windows no longer desired. This includes closing any sibling windows of a nested schema window. The user may also use the de-select capability of the right mouse button. At any level of his nested schema windows he can click his right mouse button on the most previously selected object of that particular nested schema window and watch all nested schema and sibling windows down the lineage

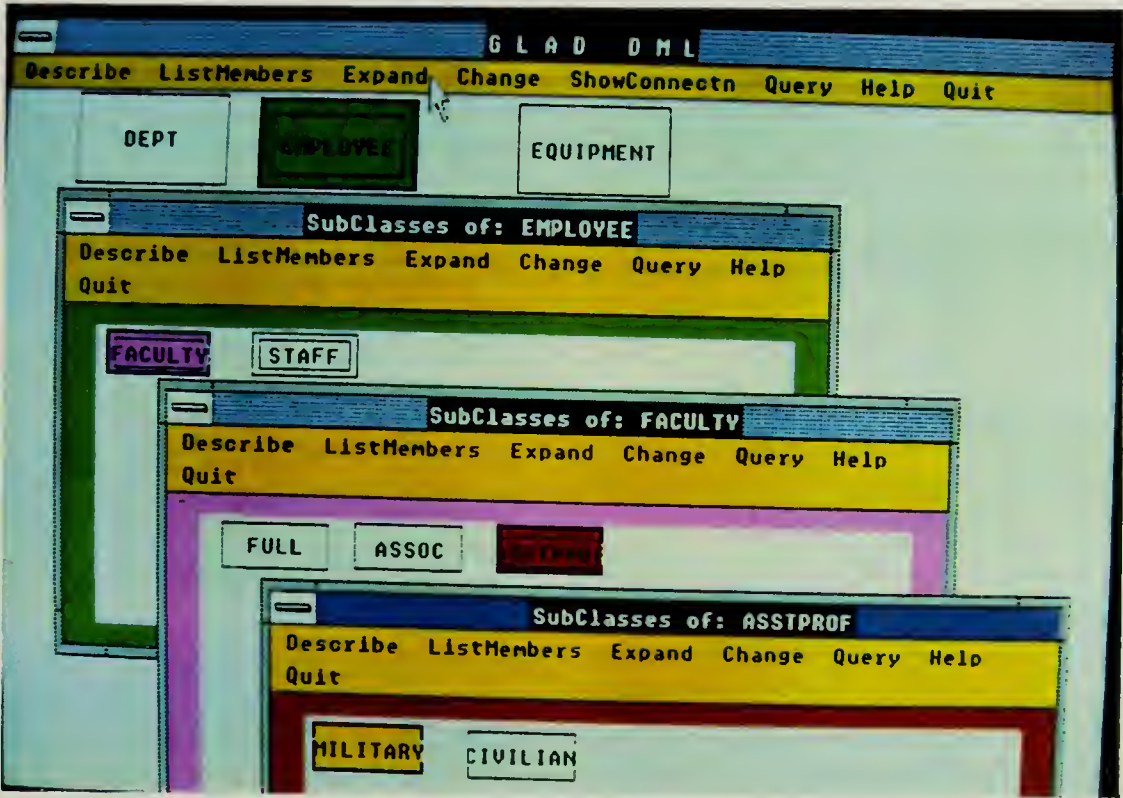


Figure 2.7 Multiple Nested DML Windows

disappear. If the desired object is not the bold line rectangle, the user need only click first with the left mouse button and follow with the right button. The user should realize by now that closing any parent or ancestor window of nested schema window, or any window object, will cause all windows created by the closed window to be destroyed.

Using Figure 2.7 as an example, the user clicks the right mouse button on the **faculty** object within the employee nested schema window. The nested schema windows for **faculty** and **asstprof** objects disappear. If the user elects to click on the *Quit* menu option of employee nested schema window, the employee, faculty and asstprof nested schema windows disappear. And since the **faculty** object is not referenced by any other object, it is restored as a non-bold line, white rectangle. We have provided the user with an option to retain or not retain the instance of the window the mouse is clicked in.

2. Data Manipulation

Now that the user knows how to view the schema and relationships within his database, we will illustrate the **GLAD** interface features that allow him to view the actual data. The user is provided with two additional windows to view his database. These two windows currently allow the user to only retrieve and view his database files.¹¹ These windows, with further development of the **GLAD** interface, will allow the user to change his data and formulate subqueries within a database. As mentioned in the initial chapter, we stop our discussion at the point where the user would be able to change database elements and/or formulate queries.

In the **DML** window's menu bar, the user need only click once on the *ListMembers* menu option and he is presented with a drop-down menu with menu options

¹¹Data files corresponding to the current schema are stored in files **employee.dat** and **dept.dat**, see Sections 2 and 3 of Appendix O.

All at Once and *One by one*. The *All at Once* menu option is for viewing all the *instances* of the selected object in the **DML** window. The number of data items viewed by the user at any given time is based on the size of the **List Members** window on the screen and where he has scrolled horizontally and vertically within the available instances of that object. The *One by one* menu option is for viewing each instance of an object one at a time with corresponding attribute names and values nicely presented in a **One Member** window. The menu option available with each of these two windows can be seen in Figure 2.1 and figures presented within this section of the chapter. The user is reminded that an object must be selected to use *All at once* and *One by one*.

This part of our **GLAD** interface development shows a very positive benefit that the use of OOL has provided us when two objects exist and we desire that they interact. These two windows are event driven and send messages to each other. Further details are discussed later in this chapter and in Chapter III.

a. All at Once

We will continue our sample session by using the **employee** object and its associated **List Members** window. The **List Members** window is created when the user clicks on the **DML** window menu option *ListMembers* and drags the mouse cursor to the first drop-down menu option of *All at once*. The employee **List Members** object reads the **employee.dat** file and shows itself on the screen.

The new window is presented to the user in the lower left quadrant of the screen, see the upper photo of Figure 2.8. We have provided the user with two visual cues to maintain an easy and quick association between the object in the **DML** window and its **List Members** window. The first cue is the object's name within the title box of the **List Members** window. Each **List Members** window receives its title from its object and can be associated easily. Secondly, which we think is easier and quicker, is that each

List Members window is created with colored border shadow within the window that corresponds to the color of its object in the **DML** window. No matter how many different windows fill the screen, the user need only see enough of the desired window and its associated color to click the mouse on it and that window is brought to the top of the screen for manipulation. For the present discussion, the employee **List Members** window is the only window open and referenced to the **employee** object. As such, the instance *5John Smith* would not be highlighted as shown in Figure 2.8. Recall that each element of data of an employee is an object and in this case is an *instance* of the **employee** object.

The menu options for the **List Members** window are show in the menu bar in the upper photo of Figure 2.8 and as a fifth level menu in Figure 2.1. Menu options *Help* and *Quit* are self-explanatory and require no preselection of an employee instance. We will not expound on the menu option *Modify* other than to say that this option is not implemented and still being developed.

Menu option *More* is implemented and provides the user with an additional window for viewing a particular employee instance in more detail. As mentioned previously, the user may have to scroll along the horizontal scroll bar to see all the attributes of an employee. Some attributes may require a large portion of a line within the **List Members** window if the user is to see it all. We developed the **List Members** window so the user can view all the attributes quickly and with enough detail to get most of the information he has stored. As the **List Members** window loads the members of the selected object, it read the size of each attribute field. Any particular attribute item that does not fill its field completely is concatenated with spaces. The maximum length any attribute field can be is twenty characters. Any attribute item that is longer than twenty characters is cut off at the sixteenth character and concatenated with three periods and one space (...). In our sample in the upper photo of Figure 2.8,

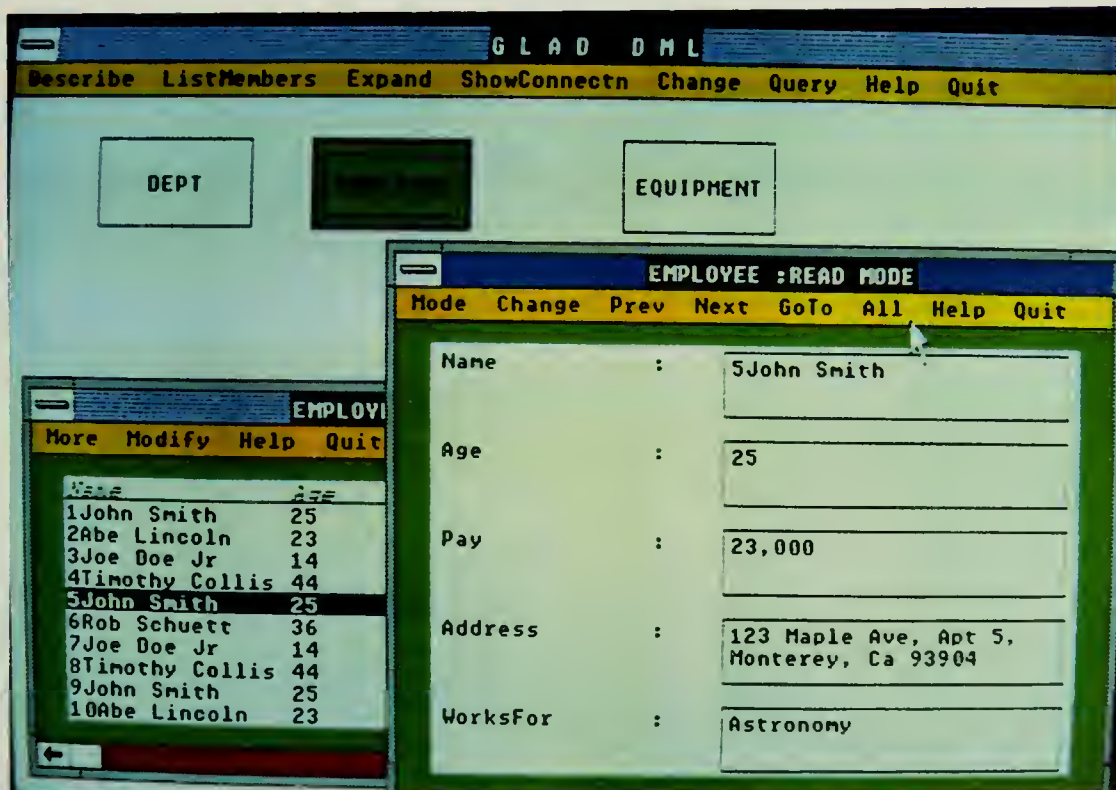
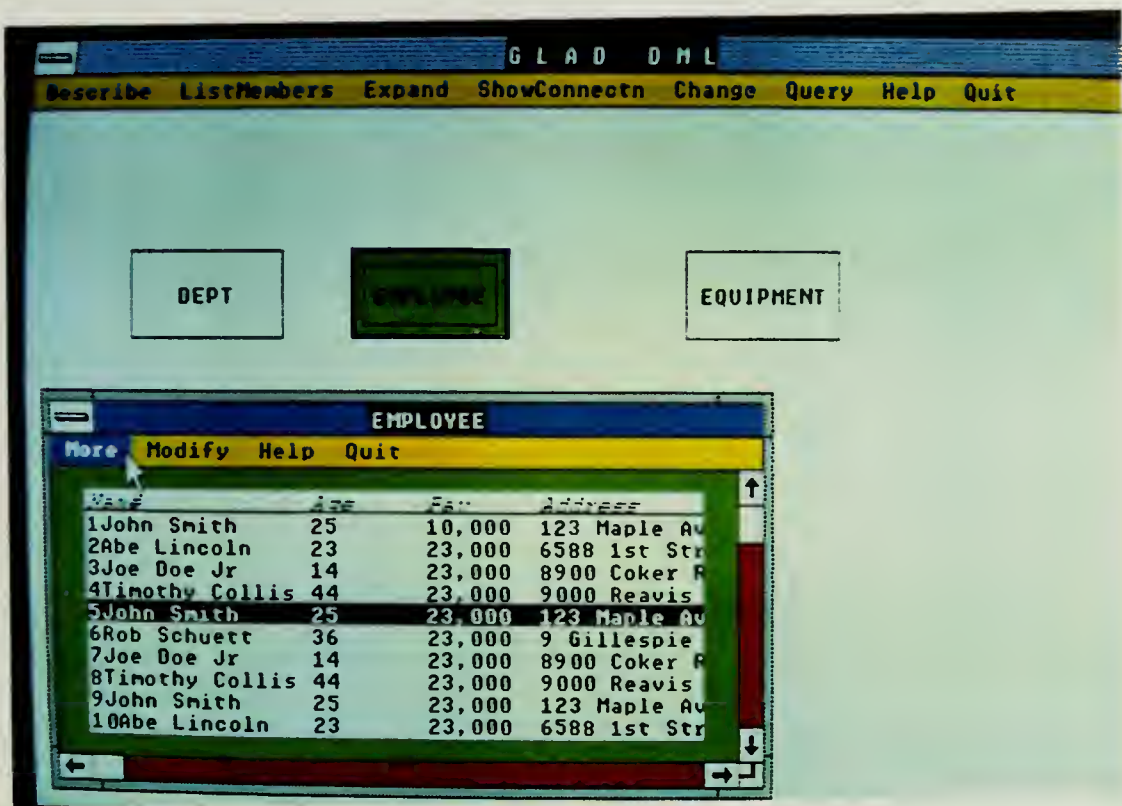


Figure 2.8 LstMemWindow and OneMemWindow Windows

the user could view the whole window horizontally by expanding the **List Members** window horizontal, a MS-Window function. The user would see

5John Smith 25 23,000 123 Maple Ave, A... Astronomy

To present the complete address of the individual would require an excess of twenty characters. The user can easily see that the address field of employee is not completely displayed because it has '...' at the end of the field.

If the user wants to initiate a menu option on a particular employee, he clicks on the line within the **List Members** window that shows the desired employee. This will highlight the data line in reverse video (see Figure 2.8). If the user changes his mind and clicks on a different line, the prior line is no longer highlighted and the new line is highlighted. If the user holds down the left mouse button, the mouse cursor can be dragged up and down the **List Members** window with a corresponding change of highlighted line within the window. The user can also de-select a highlighted line with the right mouse button. However, this feature only works if the line is highlighted from a previous selection. Using the right button on a non-highlighted line has no effect within the **List Members** window.

If the user wants more detail on a particular employee, he ensures that the employee's line is highlighted in the window and clicks the *More* menu option. A **One Member** window appears in the right portion of the screen. This is the same window the user sees if he clicks the menu option *One by one* from the **DML** window. If the user failed to select a line prior to clicking on *More*, an Error Box stating, *There is no highlighted line*, is presented to the user. Further discussion of **List Members** and **One Member** window interaction is continued after the next section on "one by one".

b. One by One

For our initial discussion and sample session of a **One Member** window, we will continue using our **employee** object from the **DML** window. The window is created when the user clicks on the **DML** window menu option *ListMemebers* and drags the mouse cursor to the second menu option *One by one*. The employee **One Member** window object reads the **employee.dat** file and shows itself on the screen.

The **One Member** window is presented to the user in the right portion of the screen, see the lower photo of Figure 2.8. Again, it is presently the only window opened and referenced to the **employee** object. As such, the first instance of employee would be seen in the **One Member** window, not the fifth instance as shown in the lower photo of Figure 2.8. The menu options can be seen in the menu bar of the lower photo of Figure 2.8 and as a fourth level menu in Figure 2.1. Menu options *Help* and *Quit* are the same. *Mode* and *Change* options are still being developed and will provide the user the ability to query and modify his database. Presently they activate drop-down menu options of *ReadOnly-Edit-Query* and *AddData-DeleteData-ModifyData* respectively (see Figure 2.2).

Menu options *Prev* and *Next* allow the user to view each employee's data by clicking back or forward one at a time through the database. These options are nice if the user wishes to view all the employees one after another or in close proximity to each other within the order of the data file. Several quick clicks on either option can move the user through several small blocks of data equivalent to the number of clicks. This is not very user friendly for large jumps within the file. Should the user try to click inappropriately on one of these options when he is at the beginning or end of file, he will be assisted with a **WAIT** dialog box stating *No more previous data* or *No more next data*, as appropriate.

If the user needs to move to the beginning, end, or a particular index of an employee of the file and clicking excessively on *Prev* or *Next* is inefficient and not user-friendly, we have provided the user with menu option *GoTo*. This option provides a drop-down menu of three sub-options that are more efficient and effective for specific moves within the database. Clicking on *GoTo* presents the options of *First*, *Last* and *Ith*, see Figure 2.2. Dragging to and releasing of the mouse cursor on option *First* will immediately display the first employee's data within the window and appropriately, *Last* displays the last employee's data.

The *Ith* option presents our first **InputDialog** box to the user. After the user selects this option, a dialog box will appear with "GoTo" in the caption bar and "type in the position# of desired data" as the prompt information. The user sees a flashing cursor bar within the text input section of the dialog box and two buttons, *OK* and *Cancel*. In our **One Member** window example, the lower photo of Figure 2.8, the user had input '5' and clicked on the *OK* button. The dialog window disappeared and the **One Member** window was immediately updated with our fifth data element, *5John Smith*. If the user tries to input a position number less than one or greater than the number of members in the file, he receives an Error Box stating *Out of range, Must be in 1..<size>*, where 'size' is the actual number of members in the file.

The last menu option of **One Member** window to discuss is *All*. If the user wants general details on the overall database, he clicks on *All* menu option. A **List Members** window appears in the lower left quadrant of the screen. This is the same window the user sees if he clicks the menu option *All at once* in the **DML** window. If there was already an instance of an employee **List Members** window, the *All* option causes that window to be called to the top view within the screen. The interaction of these two windows is now discussed in the next section.

c. LstMemWindow and OneMemWindow Interaction

The previous discussion was about a user's interface with an individual **List Members** or **One Member** window. These two windows have the capability of having the other window created or presented to the top of the screen if already in existence. When the two window are present, the user sees them as depicted in the lower photo of Figure 2.8. Depending on which window is the last one created, we see that window overlapping its sibling window.

As mentioned at the end of the section on *All at once*, the user need only select an particular instance of an **employee** object and then click on the *More* menu option within the **List Members** window. When the user releases the left mouse button while the *More* is highlighted (see the upper photo of Figure 2.8), he is presented with an associated **One Member** window (see the lower photo of Figure 2.8). The association between **employee** object, **List Members** window and **One Member** window is easily seen because of the object's name in the two windows and their common reference color, e.g., green in the lower photo of Figure 2.8. This two window setup gives the user general and specific access to his files.

In the lower photo of Figure 2.8, the user has highlighted the employee named **John Smith**. With the **One Member** window open, it shows all the data, as (sub)objects, associated with **John Smith**. When **List Members** window calls for a **One Member** window, the information of the highlighted line in our **List Members** window is depicted in the **One Member** window. This behavior is slightly different than the **DML** window's initial *One by one* reference to the first data element of the file.

If the sequence of window calls is switched and the user selects *All* option from his **One Member** window, we would see the lower photo of Figure 2.8 with the **List Members** window overlapping the **One Member** window in the same position and exact

same states. The current data element in the **One Member** window is correspondingly highlighted in the **List Members** window when it is created. In the lower photo of Figure 2.8, this data element happens to be one of the first ten elements of the data file and visible within the **List Members** window. If the user had element number '12' in his **One Member** window and initiated the *All* option, the element would be highlighted but not visible within the **List Members** window. The user would have to click on the vertical scroll bar until the twelfth element was brought into view. At that time, he would see the twelfth element highlighted within the window.

This is just the initial part of cross referencing between these two windows. When the user clicks on the *Prev*, *Next* or *GoTo* menu option, he sees the data change in his **One Member** window and a corresponding change of highlighted line in the **List Members** window, if the element is visible in the window. When the user clicks or drags the mouse within the **List Members** window, he sees the highlighted line change and a corresponding change of data in the **One Member** window. When the user deselects the highlighted line in the **List Members** window, the **One Member** window remains set to the last referenced data element until such time as one of the previously mentioned menu options is reselected or a line highlighted.

We feel that establishing this behavior between the two windows is beneficial to the user. He can see as much or as little of his data file as he wants at any given time during his manipulation of the data. This cross reference builds and maintains confidence in the user that what he sees in either or both windows is what he really wants. As the query and change data options are developed within these windows, the user will see and acknowledge this benefit.

III. IMPLEMENTATION DETAILS

An object-oriented language, particularly Actor, is an excellent high-level language for developing our **GLAD** interface. The capabilities that were expanded on in this research were powerful and will aid in developing an interface that is friendly to all users, experienced or not. This chapter presents the benefits and conclusions reached as a results of this thesis research.

A. MODULAR CONSTRUCTION

The concept of programming modularity is supported very well by object-oriented programming and Actor. Each window function — that is logical function, is implemented by a separate window class. Our top-level functions are implemented by **GladWindow.Cls**, **DML** is implemented by **DMWindow.Cls**, **Describe** is implemented by **DscrWin.Cls** and so forth. Each set of functions is handled individually in the application. A clear division of labor by classes helps us and other programmers develop a fine structured, modular application that has excellent flexibility for modification and extension in any continual development or future maintenance.

An excellent example of the need to extend and modify during development of **GLAD** was our nested schema window. The **Describe**, **List Members** and **One Member** windows were already developed when we proceeded to the nested schema window. We defined a new window class called **NestDMWind.Cls** (see Appendix H). This was easy since we made it a sub-class of our **DMWindow.Cls**, which is discussed later in this chapter. This extension to the application had to be intergrated with our existing classes.

Some time was taken in thinking along object-oriented programming verses procedural practices. This was worth the time because the actual coding was very easy.

We only had to extend the capabilities of two classes and modify two methods within one of those classes. The first extension was to add one instance variable to our current variables in **GladObj.Cls**, as shown below:

```
name
pt /*origin point of the box*/
color /*to fill the box when selected*/
nesting /*if true, it is a generalized object*/
attributes /*collection of name, class,
            and type(U or S)*/
refCnt /*reference count*/
thickBorder /*true if most recently
            selected object*/
memeberFile /*contains tuple*/
aDscrbWin /*its describe window
aLMWinn /*its list member window*/
aOMWin /*its one member window*/
aNDMWin /*its nested DML window*/
```

The **aNDMWin** variable was all that was needed to store the creation of any nested schema window from a selected **GladObj** object. The second extension was to add to our **DMWindow.Cls** a method to create a new nested schema window, see **Def expand** Appendix D, and a one line method to count the number of open nested schema windows so we can offset their position as they are created on the screen.

The modification was also simple because we only added some code to two methods in our **DMWindow.Cls**. In **Def enddrag**, we only added an **or** check:

```
if not(prevObj.aDscrbWin or prevObj.aLMWin
      or prevObj.aOMWin or prevObj.aNDMWin
```

to the already existing check for no other reference to the selected object in the DML window so it may be changed back to white. In the de-selection of the object in the DML window, **Def WM_RBUTTONDOWN**, we add an **if** statement:


```

if selObj.aNDMWin
    Call DestroyWindow(handle(selObj.aNDMWin))
endif;

```

to destroy the nested schema window should its object be clicked with the right mouse button.

B. BENEFITS OF INHERITANCE

A benefit of Actor and **GLAD** is the fact that the behavior of classes and objects can be extended or modified. A subclass allows the behavior of a parent class to be extended, which uses the principle of inheritance. Thus, the behavior of a nested schema object is an *extension* of the behavior of a **DML** object. Class methods are synonymous with an object's behavior. A **DML** method, once written, can be used by any new class that is a subclass of a **DML** Class — it is never necessary to rewrite the method. The availability of inheritance in an OOL increases the speed of the designer and increases the power of the application packet he is designing. In our design of nested schema Class we only needed to modify the behavior of our **DML** Class slightly to get the necessary results for our **GLAD** interface. Hence, only a few methods were added to create a window object that had the capability of our **DML** window minus the behavior of *ShowConnectn* method and the modification of *paint* method.

The definition of a method in a subclass overrides any definition of that method that may exist in any ancestor classes. For example, since the *paint* method in our **DML** Class was not appropriate for our nested schema Class, the method was redefined as:

```

Def paint(self,hdc)
{
    shadeOuterRegion(self,hdc);
    paint(self:DMWindow,hdc)
}!!

```

Thus, when nested schema *show* method sends a *paint* message to the above **paint**

method, the method only modifies the behavior of our new window object with a colored border around the window that corresponds to the color of the selected object within the parent window. With the second *paint* instruction within the above method, the nested schema window inherits the setup and display behaviors of its parent **DML** window.

The use of an OOL requires less duplication of code and structure than other procedural languages like Pascal and C. As the small example above demonstrated, we can abstract many common features through this inheritance capability. Our **GLAD** windows are extensions of Actor's Window Class which has inherited much of the behavior of MS-Windows. This would have to be completely specified if we had written our application in the C language.

C. BENEFITS OF MESSAGE PASSING

Message passing has made it easier when it comes to extensibility of the program and interactions between parent/child objects and sibling objects. **List Members** and **One Member** windows were developed separately. Once we realized that they should cross reference each other, it was easy to write a few new methods to handle this new requirement. We wrote the following methods so the **List Members** window and the **One Member** window would interact and cross reference each other.

```
Def hiLiteNewItem(self,newIdx)
Def displayNewMem(self,idx)
```

This required very few and very minor changes to the existing methods. In other words, if we need new functionality, which is normal for OOL, we can achieve it by creating appropriate message sending protocols between desired classes.

The benefit of message passing and letting Actor take care of executing the correct method allows the programmer to maintain a more general code. This general code can act on many classes because each object has the code explained internally or within the

ancestry of classes above it. The programmer sends the message to different objects that look to see if it has a method with the same name that can execute the instruction. This continues the different strategy of OOL that names (variable and parameters) are not typed and can be bound by any object. [Refs. 1, 6]

The example in [Ref. 5:p 33] using *print* message is a good example of this flexibility. The discussion uses the examples below:

```
print(15);  
print("Hello");
```

The *print* message is being sent to an object. In the *print*(15) case the receiver of the message is an instance of **Int** Class and in the case of *print*("Hello") the receiver is an instance of **String** Class. The same message can be sent in each case with different **print** methods handling the execution. The authors of the examples referred to this quality as *polymorphism*[Ref. 5:p. 33], the benefits of which are discussed more in the next section of this thesis.

D. BENEFITS OF POLYMORPHISM

Polymorphism in an OOL is a more efficient means of handling logic coding associated with other procedural languages. This principle is supported by the ability of our application to use one message to instruct many different objects. Objects are inherently capable of deciding how to handle a given message sent to them. A good example occurred when we were developing **List Members** and **One Member** windows.

During the implementation of cross referencing capabilities between **List Members** and **One Member** windows, there was an error invoked when there was no open **List Members** window. This could have been solved by writing the following procedural statement and adding it to the existing methods within **One Member** class:

```

if selObject.aLMWin <> nil then
    hiLiteNewMem(selObject.aLMWin, selMemIdx)
endif

```

Instead, we solved the problem by adding the method *hiLiteNewMem* into Actor's Nil Class and left the code in **One Member** class intact.

Another example is we designed each of our **GLAD** windows to behave with certain characteristics of size, with color border or without, and different menu options. Yet, our windows, **Describe**, **Show Connection**, **List Members**, **One Member** and nested schema, can receive the same *start* message from our **DML** window, as shown below:

```

start(selObj.aDscrbWin,self)

start(aCOWin,self,dbSchema)

start(selObj.aLMWin,self,selObj,
      currentSelMemIdx(self,selObj.aOMWin))

start(selObj.aOMWin,self,selObj,
      currentSelIdx(self,selObj.aLMWin))

start(selObj.aNDMWin,selObj.nesting,
      selObj.color,colorTable)

```

Each window object is capable of handling this message the way we have designed them.

A third example of the power of polymorphism in OOL is the previously mentioned *paint* method. We were able to send the *paint* message one after the other and not worry about it being handled properly by the appropriate object.

E. DISCUSSION OF RELATIONAL DATA MODEL

Since more non-computer trained personnel want to use database systems, we feel **GLAD** adds to the highest level of abstraction, the view level,¹² which the relational data model can not express when it deals with complex real world relationships. Users of the relational data model must use a collection of tables (*relations*) with unique names and rows (*tuples*) of values that represent a relationship among a set of values [Ref. 7]. Only atomic type values such as integers, reals and characters can be directly represented in the tables. The relational data model can not directly represent the complex types of data(ie. worksfor) that we can represented by using OOL.

Additional tables must be constructed and then the user must utilize cumbersome and indirect methods to try and view a complex, non-atomic type of data. The user must be familiar with constructs such as range, attributes and conditions and know logical operations and tuple calculus to master data manipulation within a relational data model. The user must rely on a *functional dependency*, the generalization of the notion of a *key*, to obtain some highly desirable normal form from his relational database [Ref. 7]. The user of a large real world database must deal with: [Ref. 7]

- determining all the functional dependencies that hold.
- choosing a particular decomposition of a relational schema.
- determining those functional dependencies that hold on the decomposed schema.
- guarding against decompositions that have abnormal behaviors similar to Korth's discussion of pitfalls in relational database designing.

Even extensions like GEM [Ref. 8] that are powerful and allow some direct expression of non-atomic types, are not user friendly to the unsophisticated and naive users.

¹²This is the highest level of abstraction a database may be viewed of the three levels of: physical, conceptual and view.

The interface between **GLAD** and the user provides a more meaningful and easier understanding of his entities, atomic and non-atomic, as objects instead of having to deal with tables, attributes and values. We feel the user can concentrate on the general structure of a non-atomic object rather than the low-level implementation details.

F. CONCLUSIONS

The use of Actor for this thesis has shown many advantages and strengths in using an OOL for data manipulation interaction. Actor provides an excellent user interface that facilitates faster development of an interactive application prototype. It did take some time to learn the Actor language and learn to think about object-oriented programming concepts and not slip into procedural methods when dealing with the development of the application. The following conclusions are presented about the use of an OOL, particularly Actor, as a result of our research.

- The available predefined classes in Actor provide robust capacity to build quick prototypes.
- The use of OOL and its encapsulation of data and manipulation of that data into one entity, an object, enforces the worthwhile principle of Information Hiding. This is done by having a narrow interface between different classes and limiting the exchange of implementation details.
- Actor and OOL support the concept of program modularity. This is done by forcing the programmer to develop his portion of code according to class association and the narrow interface different objects have between each other.
- The use of inheritance by OOL increases the power and speed of development of an application package. General features are inherited which reduces the amount of required code and structure.
- The ability to simulate real world behavior of different abstractions responding to the same general instruction is a key benefit of OOL. This concept of polymorphism relates to our human expectation that different objects or things should respond properly and correctly to a general instruction. An object is inherently capable of handling the specific details of the instruction.

The use of an object-oriented language in future enhancements to **GLAD** should provide more interesting results.

APPENDIX A - GLADAPP.CLS FILE

```
/*authorship: Wu, C.T.*/
/*Glad application*/!!

inherit(Object, #GladApp, #(display /*GLAD main window*/
), 2, nil)!!

now(GladAppClass)!!

now(GladApp)!!

Def start(self | openDlg)
{
  display :=
    new(GladWindow, ThePort, "GladTopmenu", "G L A D",
      rect(-5, -5,
        x(screenSize()),
        y(screenSize()) ) );
  show(display, 1);
  openDlg := new(Dialog);
  runModal(openDlg, ABOUT_GLAD, display)
}  !!
```

APPENDIX B - GLADWIND.CLS FILE

```
/*authorship: Wu, C.T.*/
/*display window for GLAD*/!!

inherit(Window, #GladWindow, #(dbList/*DBDialog*/
dDWin /*window for data definition*/
dMWin /*window for data manipulation*/), 2, nil)!!

now(GladWindowClass)!!

now(GladWindow)!!

Def removeDb(self)
{
  if not(dbList) /*not opened yet*/
    dbList := new(DBDialog)
  endif;

  dbList.state := REMOVE_DB;
  runModal(dbList,RMVDBLIST,self)
} !!

Def openDb(self )
{
  if not(dbList) /*not opened yet*/
    dbList := new(DBDialog)
  endif;

  dbList.state := OPEN_DB;
  if runModal(dbList,OPNDBLIST,self) == OPEN_DB

    dMWin := new(DMWindow, self, "GladDmlMenu",
      "G L A D  D M L",self.locRect);
    loadSchema(dMWin, fileNameOfSelDb(dbList));
    show(dMWin,1)
  endif
}      !!

Def topHelp(self)
{
  errorBox("Help", "will provide an aid")
}  !!
```

```

Def makeNewDb(self)
{
  dDWin := new(DDWindow,self,"GladDdlMenu",
    "G L A D  D D L",self.locRect);
  show(dDWin,1)
}    !!

Def command(self, wP, lP)
{
  select
    case lP <> 0
      is ^0
    endCase

    case wP == MAKE_NEWDB
      is makeNewDb(self)
    endCase

    case wP == OPEN_DB
      is openDb(self)
    endCase

    case wP == REMOVE_DB
      is removeDb(self)
    endCase

    case wP == TOPHELP
      is topHelp(self)
    endCase

    case wP == QUIT_GLAD
      is
        if dbList /*something is loaded*/
          updateDbsFile(dbList)
        endif;
        close(self)
    endCase

  endSelect;
  ^0
}    !!

```

APPENDIX C - DBDIALOG.CLS FILE

```
/*authorship: Wu, C.T.*/  
/*This dialog list the databases currently available  
under GLAD. The dialog uses either OPNDBLIST or  
RMVDBLIST dialog template depending on the state.  
Buttons HELP, OPEN, REMOVE and CANCEL are self  
explanatory. ABOUT describes the selected database.*!!
```

```
inherit(Dialog, #DBDialog, #(dbNames  
  /*collection of databases names  
  used for listing purpose*/  
rmvDbNames  
  /*collection of databases  
  selected to be removed*/  
state  
  /*tells whether the dialog is  
  in Open or Remove mode*/  
selDb  
  /*selected db to be opened*/), 2, nil)!!
```

```
now(DBDialogClass)!!
```

```
Def new(self | theDlg, gladDbs, line)  
{  
  theDlg := new(self:Behavior);  
  theDlg.dbNames := new(SortedCollection,15);  
  theDlg.rmvDbNames := new(SortedCollection,15);  
  theDlg.selDb := new(String,30);
```

```
  gladDbs := new(TextFile);  
  setName(gladDbs, "glad.dbs");  
  open(gladDbs,0); /*0 means read-only*/
```

```
  loop while (line := readLine(gladDbs)) begin  
    add(theDlg.dbNames, line)  
  endLoop;
```

```
  close(gladDbs);  
  ^theDlg  
} !!
```

```
now(DBDialog)!!
```



```

/*gets the filename from the name listed in the listBox*/
Def fileNameOfSelDb(self | tmpStr)
{
    tmpStr := new(String,30);
    tmpStr := "";
    do(selDb, {using(elem)
        if elem <> ' '
            tmpStr := tmpStr + asString(elem)
        endif });
    ^subString(tmpStr,0,7) + ".sch"
}    !!

Def getSelDb(self)
{
    ^selDb
}    !!

/*if confirmed, then remove the selected db from
dbNames and temporarily save it in rmvDbNames*/
Def rmvDbFrom(self ,text)
{
    if questionBox("Are you sure?",
        "Really remove"+CR_LF+text) == IDYES
        add(rmvDbNames, text);
        remove(dbNames, text)
    endif
}    !!

/*update the glad.dbs file if any db is removed*/
Def updateDbsFile(self | gladDbs)
{
    if rmvDbNames /*some db is removed*/
        do(rmvDbNames,
            {using(elem | result)
                result :=
                    questionBox("W A R N I N G",
                        "Really remove " + elem + "?");
                if result == IDNO /*then restore it*/
                    add(dbNames,elem)
                endif
            } )
        endif;

    gladDbs := new(TextFile);
    setName(gladDbs,"glad.dbs");
    create(gladDbs);
    do(dbNames, {using(elem) write(gladDbs,elem+CR_LF)});
    close(gladDbs)
}    !!

```

```

/*set the current selection to idx*/
Def getSelIdx(self)
{
    ^Call SendDlgItemMessage(hWnd,DB_LB,LB_GETCURSEL,0,0)
}    !!

Def command(self,wP,lP)
{
    select
        case wP == IDCANCEL
            is end(self,0)
            endCase

        case wP == ABOUT_DB
            is
                if (selDb := getLBText(self,DB_LB) ) /*not nil*/
                    errorBox("ABOUT","brief description "+selDb)
                else
                    errorBox("E R R O R!?!",
                        "Database was not selected properly")
                endif
            endCase

        case wP == HELP_LB
            is errorBox("Help","discuss other buttons");
            endCase

        /*selection was made and double-clicked
        or DEFBUTTON (either Open or Remove)
        was pressed*/
        case (wP == DB_LB and high(lP) = LBN_DBLCLK)
            or (wP == DEFBUTTON)
            is
                if (selDb := getLBText(self,DB_LB)) /*not nil*/
                    if state == REMOVE_DB
                        rmvDbFrom(self,selDb)
                    endif;
                    end(self,state)
                else
                    errorBox("E R R O R!?!",
                        "Database was not selected properly");
                    ^1
                endif
            endCase
        endSelect;
        ^1
    }    !!

```

```

/*set the current selection to idx*/
Def setCurSel(self, idx)
{
  ^Call SendDlgItemMessage(hWnd,DB_LB,LB_SETCURSEL,idx,0)
} !!

Def addString(self,aStr | ans)
{
  ans := Call SendDlgItemMessage(hWnd, DB_LB,
                                LB_ADDSTRING,0,lp(aStr));
  freeHandle(aStr);
  ^ans
} !!

/*initialize the listbox, the method is the Actor
equivalent of WM_INITDIALOG*/
Def initDialog(self,wP,lp)
{
  do (dbNames,
      {using(elem) addString(self,elem)});
  setCurSel(self,0)
} !!

```

APPENDIX D - DMWINDOW.CLS FILE

```
/*authorship:Wu, C.T. and Schuett,R.J.*/
/*GLAD Window for data manipulation interaction*/!!

inherit(Window, #DMWindow, #(schemaFName /*file name of schema*/
dbSchema /*meta data of opened db*/
nestDMWin/*a child window for all
    nested calls*/
prevObj /*previously selected
    object if any */
selObj /*currently selected
    object if any*/
colorTable /*available colors for shading*/
hDC /*display context*/
offset /*difference expressed as
    point between the origin of
    box and mouse position*/
rbuttonDn /*state of right button*/
objMoved /*true if object is dragged*/), 2, nil)!!

now(DMWindowClass)!!

now(DMWindow)!!

/*if aLMWin is open, its current selIdx is returned*/
Def currentSelIdx(self,openLMWin | idx)
{
    if openLMWin
        idx := openLMWin.selIdx
    else idx := 0;
    endif;
    ^idx;
} !!

/*if aOMWin is open, its current selMemIdx is
returned*/
Def currentSelMemIdx(self,openOMWin | idx)
{
    if openOMWin
        idx := openOMWin.selMemIdx
    endif;
    ^idx;
} !!

/*count the number of the nested windows opened*/
Def countOpnNDMWin(self)
{
    ^size(extract(dbSchema,{using(obj) obj.aNDMWin}))
} !!
```

```

/*count the number of the describe window opened*/
Def countOpnDscrbWin(self)
{
  ^size(extract(dbSchema,{ using(obj) obj.aDscrbWin}))
} !!

/*initialize the color table. this method
is called from the new method*/
Def init(self)
{
  colorTable := new(ColorTable,10);
  set(colorTable)
} !!

/*expand on a nested object*/
Def expand(self | screenSize,winCnt,offset1,
           offset2, aRect)
{
  if not(selObj) errorBox("ERROR!",
    "No object is selected")
  else
    if not(selObj.nesting) errorBox("ERROR!",
    "Selected object is not nested")
  else
    if selObj.aNDMWin
      Call BringWindowToTop(handle(selObj.aNDMWin))
    else
      /*open the nested window with matching
      border color and display it at the bottom
      right quadrant*/
      screenSize := screenSize();
      if x(screenSize) >= width(locRect)
        aRect := offset(locRect,-25,-25)
      else
        winCnt := countOpnNDMWin(self);
        offset1 := winCnt*25;
        offset2 := winCnt*25;
        aRect :=rect(asInt(x(screenSize)*0.3-offset1),
          asInt(y(screenSize)*0.4-offset2),
          asInt(x(screenSize)*0.98-offset1),
          asInt(y(screenSize)*0.98-offset2))
      endif;
      selObj.aNDMWin := new(NestDMWindow, self,
        "GladDmlMenu", "SubClasses of: "+selObj.name,
        aRect);
      start(selObj.aNDMWin,selObj.nesting,
        selObj.color, colorTable)
    endif;
  endif;
endif;
} !!

```



```

/*Draw line between Objects that are connected*/
Def showConnectn(self | aCOWin)
{
  changeMenu(self,CONNECT_OBJ,0,0,MF_DELETE);
  drawMenu(self);
  aCOWin := new(ConnObjWindow,self,"GladCOMenu",
    "SHOW CONNECTION", locRect);
  start(aCOWin,self,dbSchema);
} !!

```

```

/*open oneMemWin for the selected object*/
Def oneMember(self)
{
  if selObj
    if selObj.memberFile <> "" /*has data*/
      if selObj.aOMWin
        Call BringWindowToTop(handle(selObj.aOMWin))
      else
        selObj.aOMWin :=
          new(OneMemWindow,self,"GladOMMenu",
            selObj.name+": READ MODE",
            rect(asInt(x(screenSize()*0.34),
              asInt(y(screenSize()*0.27),
                asInt(x(screenSize()-10),
                  asInt(y(screenSize()-10)))));
        start(selObj.aOMWin,self,selObj,
          currentSelIdx(self,selObj.aLMWin))
      endif
    else
      errorBox(selObj.name, "Object has NO data yet")
    endif
  else
    errorBox("E R R O R","No object selected")
  endif
} !!

```

```

/*move the object*/
Def drag(self,wp,point | aLPt)
{
  if selObj
    objMoved := true;
    setup(self,hDC);
    eraseRect(selObj,hDC);
    aLPt := logicalPt(self,point);
    setNewOriginPt(selObj,aLPt,offset);
    display(selObj,hDC)
  endif
} !!

```

```

Def WM_RBUTTONDOWNUP(self,wp,lp | tmpObj)
{
  if not(rbuttonDn)
    ^0
  endif;
  rbuttonDn := nil;
  Call ReleaseCapture();
  tmpObj := objSelected(self,logicalPt(self,asPoint(lp)));
  if tmpObj /*an object is clicked with rbutton*/
    if tmpObj <> selObj
      if tmpObj.color = WHITE_COLOR
        errorBox("Wrong Button??",
          "Use LEFT button to select an object")
      else
        errorBox("E R R O R",
          "RIGHT button clicked object is not"+CR_LF+
          "the selected (bold-lined) object")
      endif
    else /* = selObj*/
      if selObj.aDscrbWin
        Call DestroyWindow(handle(selObj.aDscrbWin))
      endif;
      if selObj.aLMWin
        Call DestroyWindow(handle(selObj.aLMWin))
      endif;
      if selObj.aOMWin
        Call DestroyWindow(handle(selObj.aOMWin))
      endif;
      if selObj.aNDMWin
        Call DestroyWindow(handle(selObj.aNDMWin))
      endif;

      if selObj.refCnt = 0
        /*unshade it if not referenced by other objects*/
        avail(colorTable,selObj.color);
        selObj.color := WHITE_COLOR
      endif;
      /*now unselect it*/
      selObj.thickBorder := nil;
      setup(self,hDC);
      display(selObj,hDC)
    endif
  endif;
  selObj := nil;
  releaseContext(self,hDC)
} !!

```

```

Def WM_RBUTTONDOWN(self,wp,lp)
{
  if rbuttonDn
    ^0
  endif;
  rbuttonDn := true;
  Call SetCapture(hWnd);
  hDC := getContext(self)
} !!

/*list the members (ie instances) of the
selected object*/
Def listMembers(selfcurrSelMemIdx)
{
  if selObj
    if selObj.memberFile <> "" /*has data*/
      if selObj.aLMWin
        Call BringWindowToTop(handle(selObj.aLMWin))
      else
        selObj.aLMWin :=
          new(LstMemWindow,self,"GladLMMenu",selObj.name,
            rect(10,asInt(y(screenSize()))*0.45),
              asInt(x(screenSize()))*0.65),
              asInt(y(screenSize()))*0.95));
        start(selObj.aLMWin,self,selObj,
          currentSelMemIdx(self,selObj.aOMWin))
      endif
    else
      errorBox(selObj.name,
        "Object has NO data yet")
    endif
  else
    errorBox("E R R O R","No object selected")
  endif
} !!

/*list the members (ie instances) of the
selected object*/
Def query(self)
{
  if selObj
    errorBox("Query Box for",selObj.name)
  else
    errorBox("E R R O R","No object selected")
  endif
} !!

```

```

/*describes other DML commands*/
Def help(self)
{
  errorBox("H E L P","at your service")
} !!

/*describe the structure of the selected object*/
Def describe(self | hdc,hBrush,winCnt,screenSize,
             offset1,offset2)
{
  if selObj
    /*already has the describe window opened?
     if yes, then bring it to the top*/
    if selObj.aDscrbWin
      Call BringWindowToTop(selObj.aDscrbWin)
    else
      /*open the describe window with the matching
       border color and display it at the bottom
       righthand corner*/
      screenSize := screenSize();
      winCnt := countOpnDscrbWin(self);
      offset1:= winCnt*25;
      offset2:= winCnt*15;
      selObj.aDscrbWin :=
        new(DscrbWindow,self,nil,
          "Describe: "+selObj.name,
          rect(asInt(x(screenSize)*0.58-offset1),
            asInt(y(screenSize)*0.6 -offset2),
            asInt(x(screenSize)*0.98-offset1),
            asInt(y(screenSize)*0.97-offset2) ) );
      start(selObj.aDscrbWin,self)
    endif
  else
    errorBox("E R R O R","No object selected")
  endif
} !!

```

```

Def command(self, wP, lP)
{
  select
    case lP <> 0
      is ^0
    endCase

    case wP == DESCRIBE_OBJ
      is describe(self)
    endCase

    case wP == LIST_MEM
      is listMembers(self)
    endCase

```

```

case wP == ONE_MEM
  is oneMember(self)
endCase

case wP == CATEGORY_OBJ
  is expand(self)
endCase

case wP == CONNECT_OBJ
  is showConnectn(self)
endCase

case wP == QUERY
  is query(self)
endCase

case wP == DMHELP
  is help(self)
endCase

case wP == QUIT_DML
  is close(self)
endCase

endSelect;
} !!

/*converts the client coordinate pt to a logical pt*/
Def logicalPt(self,aClientPt | aLogiPt,aRect,wd,ht)
{
  aRect := clientRect(self);
  wd := width(aRect);
  ht := height(aRect);
  aLogiPt := new(Point);
  aLogiPt.x := aClientPt.x * 1024 /wd;
  aLogiPt.y := aClientPt.y * 512 /ht;
  ^aLogiPt
} !!

/*setup the display context*/
Def setup(self, hdc | aRect, wd, ht)
{
  Call SetMapMode(hdc,MM_ANISOTROPIC);
  aRect := clientRect(self);
  wd := width(aRect);
  ht := height(aRect);
  Call SetWindowExt(hdc,1024,512);
  Call SetViewportExt(hdc,wd,ht)
} !!

```



```

/*left button is released*/
Def endDrag(self,wp,point lALPt)
{
  select
  case selObj and not(objMoved)
  /*an object is selected and was not moved*/
  is
    if prevObj /*unbold the bolded border*/
      if not(prevObj.aDscrWin or prevObj.aLMWin
        or prevObj.aOMWin or prevObj.aNDMWin)
        and prevObj.refCnt=0
        /*also unshade since it has no DWin,OMWin,LMWin,
          NDMWin and is not referenced by other objects*/
        avail(colorTable,prevObj.color);
        prevObj.color := WHITE_COLOR
      endif;
      prevObj.thickBorder:= nil;
      display(prevObj,hDC)
    endif;
    if selObj.color = WHITE_COLOR
      /*not referenced in another's describe window,
        so assign it a color*/
      selObj.color := nextBrushColor(colorTable)
    endif;
    selObj.thickBorder := true;
    display(selObj,hDC)
  endCase

  case selObj and objMoved
  /*an object is just moved, so unselect it*/
  is
    display(selObj,hDC);
    selObj := prevObj
  endCase
endSelect;
releaseContext(self,hDC)
} !!

```

/*left button is pressed; check if the cursor is within the object rectangle. If yes get ready to move or select it*/

```

Def beginDrag(self,wp,point lALPt)
{
  aLPt := logicalPt(self,point);
  objMoved := nil;
  if selObj
    /*remember it if some object is currently selected*/
    prevObj := selObj
  endif;

```

```

if (selObj := objSelected(self,aLPt))
  offset := getOffset(selObj,aLPt)
endif;
hDC := getContext(self);
setup(self,hDC)
} !!

/*detects whether the cursor is in the object rect*/
/*everything is in a logical coordinate*/
Def objSelected(self,cursorPt)
{
  do (dbSchema,{using(obj)
    if containedIn(obj,cursorPt)
      ^obj /*return the selected obj*/
    endif });
  ^nil
} !!

/*draws the diagram*/
/*this paint method is called by the show method
via update method which sends WM_PAINT*/
Def paint(self, hdc)
{
  /*set the mode*/
  setup(self,hdc);
  /*display objects*/
  do (dbSchema, {using(obj) display(obj,hdc)})
} !!

/*gets the meta data of db to be opened
and initialize other instance variables*/
Def loadSchema(self, aSchemaFile | aFile)
{
  schemaFName := aSchemaFile;
  aFile := new(DBSchemaFile);
  setName(aFile,aSchemaFile);
  open(aFile,0); /*read-only*/
  dbSchema := getSchema(aFile);
  close(aFile)
} !!

```

APPENDIX E - DSCRBBWIN.CLS FILE

```
/*authorship: Wu, C.T. and Schuett, R.J.*/
/*window for describing the structure of a selected object*/!!

inherit(Window, #DscrbbWindow, #(aDMWin /*a DML window who called
    this describe window*/
describeObj /*object described by this window*/
toBeUnShadedObj /*collection of user defined attr*/), 2, nil)!!

now(DscrbbWindowClass)!!

now(DscrbbWindow)!!

/*prints out the user defined object
(i.e. dept, employee, etc)*/
Def printUserDefObj(self,hdc,attr,x,y laStr,hDC,aRect)
{
    aStr := new(String,25);
    aStr := attr[NAME] + stringOf(' ',10-size(attr[NAME]))
        + ": ";
    Call TextOut(hdc,x,y,lP(aStr),size(aStr));
    freeHandle(aStr);

    /*now print the class name with proper shading*/
    do(aDMWin.dbSchema,{ using(obj)
        if obj.name = attr[CLASS]
            /*remember this object for a possible
            unshading in the DML window
            at the WM_DESTROY time*/
            add(toBeUnShadedObj,obj);
            obj.refCnt := obj.refCnt + 1;
            if obj.color = WHITE_COLOR
                obj.color :=
                    nextBrushColor(aDMWin.colorTable)
            endif;
            /*color it in the DML window*/
            hdc := getContext(aDMWin);
            setup(aDMWin,hdc);
            display(obj,hdc);
            releaseContext(aDMWin,hdc);

            /*print the class name with shading
            in this describe window*/
            Call SetBkColor(hdc,obj.color);
            aRect := new(Rect);
            init(aRect,x+98,y-3,x+198,y+14);
            Call DrawText(hdc,lP(attr[CLASS]),-1,
                aRect,DT_VCENTER bitOr
                DT_SINGLELINE);
```

```

        freeHandle(attr[CLASS]);
        Call SetBkColor(hdc,WHITE_COLOR);
        ^0 /*return immediately*/
    endif
} )
!!

/*prints out the system defined object
(i.e. string, integer, etc)*/
Def printSysDefObj(self,hdc,attr,x,y | aStr)
{
    aStr := new(String,25);
    aStr := attr[NAME] + stringOf(' ',10-size(attr[NAME]))
        + ": " + attr[CLASS];
    Call TextOut(hdc,x,y,lp(aStr),size(aStr));
    freeHandle(aStr)
} !!

/*object's describe window is now closed,
so reflect the fact in the object inst var*/
Def WM_DESTROY(self,wp,lp | hDC)
{
    describeObj.aDscribWin := nil;

    hDC := getContext(aDMWin);
    setup(aDMWin,hDC);
    /*unshade it if not selObj, does not have its own
    listMemWin or oneMemWin open and nobody is
    referencing it*/
    if describeObj <> aDMWin.selObj and
        not(describeObj.aLMWin) and not(describeObj.aOMWin)
        and describeObj.refCnt = 0
        avail(aDMWin.colorTable,describeObj.color);
        describeObj.color := WHITE_COLOR;
        display(describeObj,hDC)
    endif;

    /*unshade the referenced objects in the
    DML window if they do not have their
    own describe, oneMemWin, or listMemWin window opened
    and not referenced by any other object*/
    do(toBeUnShadedObj,
        {using(obj)
            obj.refCnt := obj.refCnt - 1;
            if not(obj.aDscribWin) and not(describeObj.aLMWin)
                and not(describeObj.aOMWin)
                and obj.refCnt = 0 and not(obj.thickBorder)
                avail(aDMWin.colorTable,obj.color);
                obj.color := WHITE_COLOR;
                display(obj,hDC)
            endif});

```



```

releaseContext(aDMWin,hDC);

toBeUnShadedObj := nil; /*reset them so they will*/
describeObj      := nil; /*be garbage collected in*/
aDMWin           := nil /*dynamic memory*/
}    !!

/*show the components and their types.
 displays the shaded box for user-defined object*/
/* currently a dummy*/
Def showAttributes(self,hdc | x, y)
{
  x := SHADE_BOR_WD+5;
  y := SHADE_BOR_HT+10;
  do(describeObj.attributes,
    {using(attr)
      if attr[USER_DEF] = "U"
        printUserDefObj(self,hdc,attr,x,y)
      else
        printSysDefObj(self,hdc,attr,x,y)
      endif;
      y := y + 15
    })
}    !!

/*shade the outer region*/
Def shadeOuterRegion(self,hdc | aRect, wd, ht, hBrush)
{
  aRect := clientRect(self);
  wd := width(aRect);
  ht := height(aRect);
  hBrush:= Call CreateSolidBrush(describeObj.color);
  init(aRect,0,0,SHADE_BOR_WD,ht);
  Call FillRect(hdc,aRect,hBrush);
  init(aRect,0,0,wd,SHADE_BOR_HT);
  Call FillRect(hdc,aRect,hBrush);
  init(aRect,0,ht-SHADE_BOR_HT,wd,ht);
  Call FillRect(hdc,aRect,hBrush);
  init(aRect,wd-SHADE_BOR_WD,0,wd,ht);
  Call FillRect(hdc,aRect,hBrush);
  Call DeleteObject(hBrush)
}    !!

/*shade the outer region and show attributes*/
Def paint(self,hdc)
{
  shadeOuterRegion(self,hdc);
  showAttributes(self,hdc)
}    !!

```



```
/*initialize the instance variable and start*/
Def start(self,dmwindow)
{
  aDMWin := dmwindow;
  describeObj := aDMWin.selObj;
  /*this value must be since selObj changes as
    more describe windows are opened*/
  toBeUnShadedObj := new(OrderedCollection,5);
  show(self,1)
}  !!
```

APPENDIX F - LSTMEMWI.CLS FILE

```

/*authorship:Schuett,R.J. and Wu, C.T.*/
/*window for listing the members of a selected object*!!

inherit(Window, #LstMemWindow, #(listMemObj /*instances of this object
    is to be listed*/
tmWidth /*char width*/
tmHeight /*char height+ext.lead*/
hDC
heading /*column heading*/
members /*instances of the obj*/
topLine /*idx to the member
    at the top of window*/
leftSetBack/*no of chars to the left of display rect*/
selIdx /*idx of highlited member*/
aDMWin /*its parent*/), 2, nil)!!

now(LstMemWindowClass)!!

now(LstMemWindow)!!

/*used by OMWindow to update highlighted line to
    correspond to indexed referenced in OMWindow*/
Def hiLiteNewMem(self,newIdx)
{
    hDC := getContext(self);
    if selIdx <> nil
        invSelMember(self,selIdx)
    endif;
    selIdx := newIdx;
    invSelMember(self,selIdx);
    shadeOuterRegion(self);
    releaseContext(self,hDC)
} !!

/*the window is closed, reflect it in DMwindow and*/
/*blank out inst. var. for garbage collection*/
Def WM_DESTROY(self,wp,lp | hDC)
{
    listMemObj.aLMWin := nil;
    hDC := getContext(aDMWin);
    setup(aDMWin,hDC);
    /*unshade the object rect if it is not selObj
        and does not have a oneMemWin or decribeWin open
        and not referenced in other describeWin*/
    if listMemObj <> aDMWin.selObj and not(listMemObj.aOMWin)
        and not(listMemObj.aDscrbWin)
        and listMemObj.refCnt = 0
        avail(aDMWin.colorTable,listMemObj.color);
    listMemObj.color := WHITE_COLOR;

```

```

    display(listMemObj,hDC)
endif;
releaseContext(aDMWin,hDC);
listMemObj := nil;
tmWidth    := nil;
tmHeight   := nil;
heading    := nil;
members    := nil;
topLine    := nil;
leftSetBack:= nil;
selIdx     := nil
}          !!

```

```

/*de-hilite the currently hilited line*/
Def WM_RBUTTONDOWN(self,wp,lp l tmpIdx)
{
    tmpIdx := idxOfSelLine(self,asPoint(lp));
    if tmpIdx and selIdx = tmpIdx
        hDC := getContext(self);
        invSelMember(self,tmpIdx);
        selIdx := nil;
        releaseContext(self,hDC)
    endif
}    !!

```

```

Def command(self, wP, lP)
{
    select
        case lP <> 0
            is ^0
        endCase

        case wP == LM_MORE
            is
                if selIdx
                    if listMemObj.aOMWin
                        Call BringWindowToTop(handle(listMemObj.aOMWin))
                    else
                        listMemObj.aOMWin := new(OneMemWindow,aDMWin,
                            "GladOMMenu",listMemObj.name+" :READ MODE",
                            rect(asInt(x(screenSize()*0.34),
                                asInt(y(screenSize()*0.27),
                                asInt(x(screenSize()-10),
                                asInt(y(screenSize()-10)))));
                        start(listMemObj.aOMWin,aDMWin,listMemObj,selIdx)
                    endif
                else
                    errorBox("WAIT","There is no highlighted line")
                endif
            endCase

```

```

case wP == LM_MODIFY
is
  if selIdx
    errorBox("Modify",asString(selIdx))
  else
    errorBox("WAIT","There is no highlited line")
  endif
endCase

case wP == LMHELP
is errorBox("Help","will help")
endCase

case wP == QUIT_LM
is close(self)
endCase

endSelect;
^0
}      !!

Def idxOfSelLine(self,pt | line, tmp)
{
  if (tmp:=pt.y-SHADE_BOR_HT-tmHeight) < 0
    ^nil
  else
    line := topLine +tmp/tmHeight;
    if line > min(size(members),topLine+visLines(self))-1
      ^nil
    else
      ^line
    endif
  endif
}  !!

Def drag(self,wp,pt | tmpIdx)
{
  tmpIdx := idxOfSelLine(self,pt);
  if tmpIdx cand tmpIdx <> selIdx
    if selIdx
      invSelMember(self,selIdx)
    endif;
    invSelMember(self,tmpIdx);
    selIdx := tmpIdx;
    displayNewMem(listMemObj.aOMWin,selIdx)
  endif
}  !!

```

```

Def invSelMember(self,idx | x0,y0)
{
  x0 := SHADE_BOR_WD;
  y0 := (idx - topLine)*tmHeight + SHADE_BOR_HT+tmHeight;
  Call PatBlt(hDC,x0,y0,width(memberRect(self)),
    tmHeight,DSTINVERT)
} !!

```

```

Def endDrag(self,wp,pt)
{
  releaseContext(self,hDC)
} !!

```

```

Def beginDrag(self,wp,pt | tmpIdx)
{
  hDC := getContext(self);
  tmpIdx := idxOfSelLine(self,pt);
  if tmpIdx
    if selIdx
      invSelMember(self,selIdx)
    endif;
  invSelMember(self,tmpIdx);
  selIdx := tmpIdx;
  displayNewMem(listMemObj.aOMWin,selIdx)
endif
} !!

```

/*window size has changed so repaint it.
topLine and leftSetBack need to be adjusted
when bottommost or rightmost page is displayed
and the window got enlarged*/

```

Def reSize(self,wp,lp)
{
  topLine := min(topLine,
    max(0,
      size(members)-visLines(self)+1));
  leftSetBack:=min(leftSetBack,
    max(0,size(members[0])
      -visColumns(self)+1));
  repaint(self)
} !!

```

```

Def WM_HSCROLL(self,wp,lp | memHeadRect, tmp)
{
  memHeadRect := inflate(clientRect(self),
    negate(SHADE_BOR_WD),
    negate(SHADE_BOR_HT));

```



```

select
case wp == SB_LINEDOWN and
  leftSetBack + visColumns(self) <= size(members[0])
is tmp:=min(leftSetBack+5,
  size(members[0])-visColumns(self)+1);
  Call ScrollWindow(hWnd,
    negate((tmp-leftSetBack)*tmWidth),
    0,0,memHeadRect);
  leftSetBack := tmp;
  setHorzScrollPos(self);
  update(self)
endCase

case wp ==SB_LINEUP and leftSetBack > 0
is tmp := max(leftSetBack-5,0);
  Call ScrollWindow(hWnd,(leftSetBack-tmp)*tmWidth,
    0,0,memHeadRect);
  leftSetBack := tmp;
  setHorzScrollPos(self);
  update(self)
endCase

case wp == SB_PAGEDOWN
is leftSetBack
  := min(leftSetBack + visColumns(self),
    size(members[0])-visColumns(self))+1;
  setHorzScrollPos(self);
  Call InvalidateRect(hWnd,memHeadRect,1)
endCase

case wp == SB_PAGEUP
is leftSetBack:= max(leftSetBack-visColumns(self),0);
  setHorzScrollPos(self);
  Call InvalidateRect(hWnd,memHeadRect,1)
endCase

case wp == SB_THUMBPOSITION
is leftSetBack := asInt(low(lp));
  setHorzScrollPos(self);
  Call InvalidateRect(hWnd,memHeadRect,1)
endCase
endSelect
)      !!

```

```

Def WM_VSCROLL(self,wp,lp)
{
  select
    case wp == SB_LINEDOWN and
      topLine <= size(members) - visLines(self)
    is topLine := topLine + 1;
      Call ScrollWindow(hWnd,0,negate(tmHeight),0,
        memberRect(self));
      setVertScrollPos(self);
      update(self)
    endCase

    case wp == SB_LINEUP and topLine > 0
    is topLine := topLine - 1;
      Call ScrollWindow(hWnd,0,tmHeight,
        0,memberRect(self));
      setVertScrollPos(self);
      update(self)
    endCase

    case wp == SB_PAGEDOWN
    is topLine := min(topLine + visLines(self),
      size(members)-visLines(self))+1;
      setVertScrollPos(self);
      Call InvalidateRect(hWnd,memberRect(self),1)
    endCase

    case wp == SB_PAGEUP
    is topLine := max(0,topLine - visLines(self));
      setVertScrollPos(self);
      Call InvalidateRect(hWnd,memberRect(self),1)
    endCase

    case wp == SB_THUMBPOSITION
    is topLine := asInt(low(lp));
      setVertScrollPos(self);
      Call InvalidateRect(hWnd,memberRect(self),1)
    endCase
  endSelect
} !!

/*returns the rect for member listing
for the purpose of invalidating*/
Def memberRect(self | aRect)
{
  aRect := new(Rect);
  ^init(aRect,SHADE_BOR_WD,SHADE_BOR_HT+tmHeight,
    width(clientRect(self)) - SHADE_BOR_WD,
    height(clientRect(self))- SHADE_BOR_HT)
} !!

```

```

Def visColumns(self)
{
  ^(width(clientRect(self)) - 2*SHADE_BOR_WD) / tmWidth
} !!

Def setVertScrollPos(self)
{
  Call SetScrollPos(hWnd,SB_VERT,topLine,1)
} !!

Def setHorzScrollPos(self)
{
  Call SetScrollPos(hWnd,SB_HORZ,leftSetBack,1)
} !!

/*print the heading in the bold face*/
Def printHeading(self | logFont,hBoldFont,hOldFont)
{
  logFont := new(Struct,50);
  putMSB(logFont,1,10); /*make it italic*/
  putLSB(logFont,1,10); /*and underlined*/
  hBoldFont := Call CreateFontIndirect(logFont);
  hOldFont := Call SelectObject(hDC,hBoldFont);
  Call TextOut(hDC,SHADE_BOR_WD - leftSetBack*tmWidth,
    SHADE_BOR_HT,IP(heading),size(heading));
  freeHandle(heading);
  Call SelectObject(hDC,hOldFont) /*restore the reg font*/
} !!

/*returns the number of lines that can appear
on the window, offsetting shadings and
heading line*/
Def visLines(self)
{
  ^(height(clientRect(self))-2*SHADE_BOR_HT-tmHeight)
  /tmHeight
} !!

Def setup(self | high)
{
  high := max(0,size(members)-visLines(self)+1);
  Call SetScrollRange(hWnd,SB_VERT,0,high,1);

  high := max(0,size(members[0])-visColumns(self)+1);
  Call SetScrollRange(hWnd,SB_HORZ,0,high,1)
} !!

```

```

/*load the members for this object*/
Def loadMembers(self | aFile, line, aRow, aStr)
{
  aFile := new(TextFile);
  setName(aFile,listMemObj.memberFile);
  open(aFile,0);
  aStr := new(String,100);
  loop while (line := readLine(aFile))
    aRow := new(String,100);
    aRow := nil;
    do (listMemObj.attributes,
      {using(attr)
        aStr := subString(line,0,indexOf(line,'&',0));
        line := delete(line,0,size(aStr)+1);
        if size(aStr) > 20
          aStr := subString(aStr,0,16)+"... "
        else
          aStr := aStr + stringOf(' ',
            min(20,asInt(attr[LENGTH],10))-size(aStr))
        endif;
        aRow := aRow + aStr });
    add(members,aRow)
  endLoop;
  close(aFile)
} !!

```

```

/*load the string value for the heading*/
Def loadHeading(self)
{
  heading := nil;
  do(listMemObj.attributes,
    {using(attr)
      heading :=
        heading + attr[NAME] +
        stringOf(' ',min(19,asInt(attr[LENGTH],10))
          - size(attr[NAME]))
    })
} !!

```

```

/*Initialize text metrics data for this window.
Load the font data into textMetrics, set the text
width and height instance variables.*/
Def initTextMetrics(self | hdc tm)
{ tm := new(Struct, 32);
  Call GetTextMetrics(hdc := Call GetDC(hWnd),
    tm);
  tmWidth := asInt(wordAt(tm, 10));
  tmHeight := asInt(wordAt(tm, 8))
  + asInt(wordAt(tm, 0));
  Call ReleaseDC(hWnd, hdc);
} !!

```



```

/*list the members of listMemObj*/
Def listMembers(self lx,y,aStr)
{
    x := SHADE_BOR_WD - leftSetBack*tmWidth;
    y := SHADE_BOR_HT + tmHeight;
    do(over(topLine,min(size(members),
        topLine+visLines(self)+1)),
        {using(idx)
            aStr := members[idx];
            Call TextOut(hDC,x,y,IP(aStr),size(aStr));
            freeHandle(aStr);
            y := y + tmHeight
        }
    )
}    !!

/*shade the outer region*/
Def shadeOuterRegion(self l aRect, wd, ht, hBrush)
{
    aRect := clientRect(self);
    wd := width(aRect);
    ht := height(aRect);
    hBrush:= Call CreateSolidBrush(listMemObj.color);
    init(aRect,0,0,SHADE_BOR_WD,ht);
    Call FillRect(hDC,aRect,hBrush);
    init(aRect,0,0,wd,SHADE_BOR_HT);
    Call FillRect(hDC,aRect,hBrush);
    init(aRect,0,ht-SHADE_BOR_HT,wd,ht);
    Call FillRect(hDC,aRect,hBrush);
    init(aRect,wd-SHADE_BOR_WD,0,wd,ht);
    Call FillRect(hDC,aRect,hBrush);
    Call DeleteObject(hBrush)
}    !!

/*shade the outer region and list the members*/
Def paint(self,hdc)
{
    hDC := hdc;
    setup(self);
    printHeading(self);
    listMembers(self);
    shadeOuterRegion(self);
    if selIdx
        invSelMember(self,selIdx)
    endif
}    !!

```



```
Def start(self, parent, obj, idx)
```

```
{  
    listMemObj := obj;  
    aDMWin := parent;  
    selIdx := idx;  
    leftSetBack:=0;  
    topLine := 0;  
    members := new(OrderedCollection,50);  
    heading := new(String,100);  
    loadHeading(self);  
    loadMembers(self);  
    initTextMetrics(self);  
    show(self,1)  
}    !!
```

```
/*add vert. and hor. scroll bars to the regular widow*/
```

```
Def create(self,par,wName,rect,style)
```

```
{  
    create(self:Window,par,wName,rect,  
        style bitOr WS_HSCROLL bitOr WS_VSCROLL)  
} !!
```

APPENDIX G - ONEMEMWI.CLS FILE

```
/*authorship:Wu, C.T. and Schuett, R.J.*/
/*a window for listing one member of an object in a long format*/!!

inherit(Window, #OneMemWindow, #(selObject /*a member of this object
    is displayed*/
selMemIdx /*idx of member to be
    displayed*/
members /*instances of selObject*/
editControl/*array of ECs*/
hDC
tmWidth /*char width*/
tmHeight /*char height*/
aDMWin /*parent DML window*/
mode /*readOnly,Edit,or Query*/), 2, nil)!!

now(OneMemWindowClass)!!

now(OneMemWindow)!!

/*user wants to select previous member*/
Def selectPrev(self)
{
    if selMemIdx == 0
        errorBox("WAIT","No more previous data")
    else
        selMemIdx := selMemIdx - 1;
        invalidateEC(self)
    endif
} !!

/*user wants to select next member*/
Def selectNext(self)
{
    if selMemIdx == size(members) -1
        errorBox("WAIT","No more next data")
    else
        selMemIdx := selMemIdx + 1;
        invalidateEC(self)
    endif
} !!
```

```

/*user wants to select lth member*/
Def selectITH(self | i, iP)
{
    iP := new(InputDialog,"GO TO",
        "Type in the position# of desired data","");
    if runModal(iP,INPUT_BOX,self) == IDOK
        i := asInt(getText(iP),10);
        if i < 1 or i > size(members)
            errorBox("ERROR","Out of Range"+
                CR_LF+"Must be in 1.."+
                asString(size(members)) )
        else
            selMemIdx := i-1;
            invalidateEC(self)
        endif
    endif
} !!

/*used by LMWindow to update OMWindow's content to
correspond to LMWindow's referenced index. Added to NilClass*/
Def displayNewMem(self,idx)
{
    selMemIdx := idx;
    invalidateEC(self)
} !!

Def invalidateEC(self | fieldCnt)
{
    fieldCnt := size(selObject.attributes);
    do( over(0,fieldCnt), {using(idx)
        setText(editControl[idx],members[selMemIdx][idx])});
    hDC := getContext(self);
    shadeOuterRegion(self);
    releaseContext(self, hDC);
    hiLiteNewMem(selObject.aLMWin,selMemIdx)
} !!

/*start or bring to top the sibling LMWindow*/
Def listMembers(self)
{
    if selObject.aLMWin
        Call BringWindowToTop(handle(selObject.aLMWin))
    else
        selObject.aLMWin := new(LstMemWindow,aDMWin,
            "GladLMMenu",selObject.name,
            rect(10,asInt(y(screenSize()*0.45),
                asInt(x(screenSize()*0.65),
                asInt(y(screenSize()*0.95))));
        start(selObject.aLMWin,aDMWin,selObject,selMemIdx)
    endif
} !!

```

```

/*the window is closed, reflect it in DMwindow and*/
/*blank out inst. var. for garbage collection*/
Def WM_DESTROY(self,wp,lp | hDC)
{
  selObject.aOMWin := nil;
  hDC := getContext(aDMWin);
  setup(aDMWin,hDC);
  /*unshade the object rect if it is not selObj,
  does not have aLMWin or describeWin open,
  and not referenced in other describeWin*/
  if selObject <> aDMWin.selObj and not(selObject.aLMWin)
    and not(selObject.aDscrWin)
    and selObject.refCnt = 0
    avail(aDMWin.colorTable,selObject.color);
    selObject.color := WHITE_COLOR;
    display(selObject,hDC)
  endif;
  releaseContext(aDMWin,hDC);
  selMemIdx := nil;
  selObject := nil;
  members := nil;
  editControl:= nil;
  tmWidth := nil;
  aDMWin := nil;
  mode := nil
}
!!

```

```

Def command(self,wp,lp | aStr)
{
  select
    case lp <> 0
    is ^0
    endCase

    case wp == READ_ONLY and mode <> READ_ONLY
    is
      unCheckMenuItem(self,mode);
      checkMenuItem(self,READ_ONLY);
      aStr := selObject.name + ":READ MODE";
      Call SetWindowText(hWnd,lp(aStr));
      freeHandle(aStr);
      mode := READ_ONLY
    endCase

    case wp == EDIT and mode <> EDIT
    is
      unCheckMenuItem(self,mode);
      checkMenuItem(self,EDIT);
      aStr := selObject.name + ":EDIT MODE";
      Call SetWindowText(hWnd,lp(aStr));
      freeHandle(aStr);

```

```

    mode := EDIT
endCase

case wp == QUERY and mode <> QUERY
is
    unCheckMenuItem(self,mode);
    checkMenuItem(self,QUERY);
    aStr := selObject.name + ":QUERY MODE";
    Call SetWindowText(hWnd,lp(aStr));
    freeHandle(aStr);
    mode := QUERY
endCase

case wp == NEXT
is
    selectNext(self)
endCase

case wp == PREV
is
    selectPrev(self)
endCase

case wp == FIRST
is
    selMemIdx := 0;
    invalidateEC(self)
endCase

case wp == LAST
is
    selMemIdx := size(members) -1;
    invalidateEC(self)
endCase

case wp == ITH
is
    selectITH(self)
endCase

case wp == LIST_MEM
is
    listMembers(self)
endCase

case wp == QUIT_OM
is
    close(self)
endCase
endSelect
}    !!

```



```

Def printEC(self,x,y,idx)
{
    setCRect(editControl[idx],
        rect(x,y,x+25*tmWidth,y+3*tmHeight));
    moveWindow(editControl[idx]);
    setText(editControl[idx],members[selMemIdx][idx]);
    show(editControl[idx],1)
} !!

Def printLabel(self,x,y,idx | aStr)
{
    aStr := selObject.attributes[idx][0];
    Call TextOut(hDC,x,y,lp(aStr),size(aStr));
    aStr := ":";
    Call TextOut(hDC,x+15*tmWidth,y,lp(aStr),size(aStr));
    freeHandle(aStr)
} !!

Def displayValues(self | idx,x,y,ecX )
{
    idx := 0;
    x := SHADE_BOR_WD +5;
    y := SHADE_BOR_HT +2;
    ecX := x + 20*tmWidth;
    do( selObject.attributes,
        {using(attr)
            printLabel(self,x,y,idx);
            printEC(self,ecX,y,idx);
            y := y + 4*tmHeight;
            idx := idx + 1 })
} !!

Def paint(self,hdc)
{
    hDC := hdc;
    displayValues(self);
    shadeOuterRegion(self)
} !!

```

```

/*shade the outer region*/
Def shadeOuterRegion(self | aRect, wd, ht, hBrush)
{
  aRect := clientRect(self);
  wd := width(aRect);
  ht := height(aRect);
  hBrush:= Call CreateSolidBrush(selObject.color);
  init(aRect,0,0,SHADE_BOR_WD,ht);
  Call FillRect(hDC,aRect,hBrush);
  init(aRect,0,0,wd,SHADE_BOR_HT);
  Call FillRect(hDC,aRect,hBrush);
  init(aRect,0,ht-SHADE_BOR_HT,wd,ht);
  Call FillRect(hDC,aRect,hBrush);
  init(aRect,wd-SHADE_BOR_WD,0,wd,ht);
  Call FillRect(hDC,aRect,hBrush);
  Call DeleteObject(hBrush)
} !!

Def createECs(self | fieldCnt)
{
  fieldCnt := size(selObject.attributes);
  editControl := new(Array,fieldCnt);
  do( over(0,fieldCnt),
    {using(idx)
      editControl[idx] := new(Edit,idx,self,
        WS_BORDER bitOr WS_CHILD bitOr ES_LEFT
        bitOr ES_MULTILINE))}
  )
  !!

Def loadMembers(self | fields, fieldCnt, line, aFile)
{
  aFile := new(TextFile);
  setName(aFile,selObject.memberFile);
  open(aFile,0);
  fieldCnt := size(selObject.attributes);
  loop while (line := readLine(aFile))
  {
    fields := new(Array,fieldCnt);
    do( over(0,fieldCnt),
      {using(idx)
        fields[idx] :=
          subString(line,0,indexOf(line,'&',0));
        line := delete(line,0,size(fields[idx])+1)});
    add(members,fields)
  }
  endLoop
} !!

```

```

/*Initialize text metrics data for this window.
  Load the font data into textMetrics, set the text
  width and height instance variables.*/
Def initTextMetrics(self | hdc tm)
{ tm := new(Struct, 32);
  Call GetTextMetrics(hdc := Call GetDC(hWnd), tm);
  tmWidth := asInt(wordAt(tm, 10));
  tmHeight := asInt(wordAt(tm, 8))
  + asInt(wordAt(tm, 0));
  Call ReleaseDC(hWnd, hdc);
} !!

Def start(self,parent,obj,idx)
{
  mode := READ_ONLY;
  checkMenuItem(self,mode);
  selObject := obj; /*need to pass obj since >1 LMWin*/
  aDMWin := parent;/*could be open*/
  selMemIdx := idx;
  members := new(OrderedCollection,50);
  initTextMetrics(self);
  loadMembers(self);
  createECs(self);
  show(self,1)
} !!

```

APPENDIX H - NESTDMWI.CLS FILE

```
/*authorship:Wu, C.T.*/
/*A nested DMWindow for displaying nested objects*/!!

inherit(DMWindow, #NestDMWindow, #(shadeColor /*for cRect border*/
), 2, nil)!!

now(NestDMWindowClass)!!

now(NestDMWindow)!!

/*This method is called by the show method via
update method*/
Def paint(self,hdc)
{
    shadeOuterRegion(self,hdc);
    paint(self:DMWindow,hdc)
} !!

/*shade the outer region*/
Def shadeOuterRegion(self,hdc | aRect, wd, ht, hBrush)
{
    aRect := clientRect(self);
    wd := width(aRect);
    ht := height(aRect);
    hBrush:= Call CreateSolidBrush(shadeColor);
    init(aRect,0,0,SHADE_BOR_WD,ht);
    Call FillRect(hdc,aRect,hBrush);
    init(aRect,0,0,wd,SHADE_BOR_HT);
    Call FillRect(hdc,aRect,hBrush);
    init(aRect,0,ht-SHADE_BOR_HT,wd,ht);
    Call FillRect(hdc,aRect,hBrush);
    init(aRect,wd-SHADE_BOR_WD,0,wd,ht);
    Call FillRect(hdc,aRect,hBrush);
    Call DeleteObject(hBrush)
} !!

/*initialize the instance variables and start*/
Def start(self,aObjColl,aShadeColor,aColorTbl | nullStr)
{
    dbSchema := aObjColl;
    shadeColor := aShadeColor;
    colorTable := aColorTbl;
    nullStr := "";
    changeMenu(self,CONNECT_OBJ, IP(nullStr),0,MF_DELETE);
    freeHandle(nullStr);
    drawMenu(self);
    show(self,1)
} !!
```

APPENDIX I - CONNOBJW.CLS FILE

```
/*authorship: Schuett, R.J.*/
/*GLAD Window to draw a line between connected objects*!!

inherit(Window, #ConnObjWindow, #(cSchema /*meta data of open db*/
cnnctColl /*collection of objects
        to be connected*/
hDC /*display context*/), 2, nil)!!

now(ConnObjWindowClass)!!

now(ConnObjWindow)!!

/*make a COPY of the schema for use, so as not to
change the original data*/
Def loadSchema(self, anObjColl)
{
    cSchema := new(OrderedCollection,15);
    do(anObjColl,{using(obj) add(cSchema, copy(obj))});
} !!

/*receives the meta data of db that was opened*/
Def start(self,dmWindow,aSchema lcnnctSchema)
{
    aDMWin := dmWindow;
    loadSchema(self, aSchema);
    cnnctColl := getConnSchema(self);
    /*no connection between objects*/
    if size(cnnctColl) = 0
        errorBox("S C H E M A",
            "No connections within this Data");
        close(self)
    else
        show(self,1)
    endif
} !!

/*the window is closed, reflect it in DMwindow and*/
/*blank out inst. var. for garbage collection*/
Def WM_DESTROY(self,wp,lp l aStr)
{
    cSchema := nil;
    cnnctColl := nil;
    aStr := "ShowConnectn";
    changeMenu(parent,QUERY,lp(aStr),CONNECT_OBJ,MF_INSERT);
    freeHandle(aStr);
    drawMenu(parent)
} !!
```



```

/*draw line between center points of obj rects*/
Def drawCnnctLine(self,hdc, attr, sPt | fPt)
{
  do(cnnctColl, {using(obj)
    if obj.name = attr[CLASS]
      fPt := point(obj.pt.x+70,obj.pt.y+38);
      line(sPt, fPt, hdc)
    endif;
  });
} !!

/*matches center of obj rect with center of connected
obj rect*/
Def connObjects(self, hdc | cntrPt)
{
  do(cnnctColl, {using(obj)
    do(obj.attributes, {using(attr)
      if attr[USER_DEF] = "U"
        cntrPt := point(obj.pt.x+70, obj.pt.y+38);
        drawCnnctLine(self,hdc,attr,cntrPt)
      endif;
    });
  });
  /*display objects*/
  do (cnnctColl, {using(obj) display(obj, hdc)})
} !!

/*sets up to ten objects for connecting*/
Def setDisplayPos(self,cntr | tmppt)
{
  select
    case cntr = 0
      is tmppt := 450@10
    endCase
    case cntr = 1
      is tmppt := 50@200
    endCase
    case cntr = 2
      is tmppt := 860@200
    endCase
    case cntr = 3
      is tmppt := 450@445
    endCase
    case cntr = 4
      is tmppt := 50@310
    endCase
    case cntr = 5
      is tmppt := 860@310
    endCase

```

```

case cntr = 6
  is tmppt := 250@70
endCase
case cntr = 7
  is tmppt := 650@70
endCase
case cntr = 8
  is tmppt := 250@390
endCase
case cntr = 9
  is tmppt := 650@390
endCase
case cntr > 9
  is tmppt := point(10, 5 + 10*(cntr-10))
endCase
endSelect;

```

```

  ^tmppt
}  !!

```

```

/*gets the schema info of the connected objects,
stored them in an ordered collection and return it*/
Def getConnSchema(self | aColl, objInColl, shouldBeInC,
                  countr)

```

```

{
  countr := 0;
  aColl := new(OrderedCollection,20);
  objInColl := new(Set, 20);
  shouldBeInC := new(Set, 20);
  do(cSchema,
    {using(obj) do(obj.attributes,
      {using(attr)
        if attr[USER_DEF] = "U"
          if not(obj.name in objInColl)
            add(aColl,obj);
            add(objInColl, obj.name)
          endif;
          if not(attr[CLASS] in shouldBeInC)
            add(shouldBeInC, attr[CLASS]);
          endif;
        endif;
      });
    });
  do(cSchema,
    {using(obj)
      if obj.name in shouldBeInC
        if not(obj.name in objInColl)
          add(aColl,obj)
        endif;
      endif;
    });
}

```

```

do(aColl, {using(obj)
  obj.pt := setDisplayPos(self, countr);
  countr := countr + 1;
});
^aColl
} !!

/*setup the display context*/
Def setup(self, hdc | aRect, wd, ht)
{
  Call SetMapMode(hdc,MM_ANISOTROPIC);
  aRect := clientRect(self);
  wd := width(aRect);
  ht := height(aRect);
  Call SetWindowExt(hdc,1024,512);
  Call SetViewportExt(hdc,wd,ht)
} !!

/*draws the diagram*/ /*this paint method is called
by the show method via update method which sends
WM_PAINT*/
Def paint(self, hdc)
{
  /*set the mode*/
  setup(self,hdc);
  /*draw lines*/
  connObjects(self, hdc)
} !!

Def command(self, wP, lP)
{
  select
    case lP <> 0
      is ^0
    endCase

    case wP == QUIT_CO
      is close(self)
    endCase
  endSelect;
  ^0
} !!

```

APPENDIX J - DBSCHEMA.CLS FILE

```
/*authorship: Wu, C.T. and Schuett, R.J.*/
/*file containing a database schema*/!!

inherit(TextFile, #DBSchemaFile, nil, 2, nil)!!

now(DBSchemaFileClass)!!

now(DBSchemaFile)!!

/*get the attributes for the object*/
Def getAttr(self | aColl, anAttr, aStr)
{
  aColl := new(OrderedCollection,10);
  loop
  while ( (aStr := readLine(self)) <> "&&" )
    anAttr := new(Array,4);
    do (over(NAME,USER_DEF+1),
      {using(idx)
        anAttr[idx] :=
          subString(aStr,0,indexOf(aStr,'&',0));
        aStr := delete(aStr,0,size(anAttr[idx])+1)});
    add(aColl,anAttr)
  endLoop;
  ^aColl
} !!

/*gets the schema info, stored them in an ordered
collection and return it*/
Def getSchema(self | schemaColl, aGladObj)
{
  schemaColl := new(OrderedCollection,10);
  loop
  while (aGladObj := nextObj(self))
    add(schemaColl,aGladObj)
  endLoop;
  ^schemaColl
} !!

/*gets the next object from the schema file*/
Def nextObj(self | tmpObj)
{
  tmpObj := new(GladObj);
  if ((tmpObj.name := readLine(self)) <> "@@")
    /*there's more*/
    tmpObj.pt := asPoint(readLine(self));
    if (at(readLine(self),0)=='G')
      tmpObj.nesting := getSchema(self)
    endif;
}
```

```
tmpObj.attributes:= getAttr(self);
tmpObj.memberFile:= readLine(self);
tmpObj.refCnt := 0;
tmpObj.color := WHITE_COLOR; /*unselected color*/
/*more fields later*/
^tmpObj
else
  ^nil
endif
)      !!
```


APPENDIX K - GLADOBJ.CLS FILE

```
/*authorship: Wu, C.T.*/
/*for storing Glad objects such as emp, dept, etc.*!!

inherit(Object, #GladObj, #(name
pt /*origin point of the box*/
color /*to fill the box when selected*/
nesting /*if true, it is a generalized object*/
attributes /*collection of name, class,
            and type(U or S)*/
refCnt /*reference count*/
thickBorder/*true if most recently
            selected object*/
memberFile/*contains tuple*/
aDscrWin /*its describe window*/
aLMWin /*its listMemWindow*/
aOMWin /*its oneMemWidow*/
aNDMWin /*its nestedDMWindow*/), 2, nil)!!

now(GladObjClass)!!

now(GladObj)!!

/*returns an inner box for a generalized object*/
Def nestedRect(self | aRect)
{
  aRect := new(Rect);
  aRect := rect(self);
  ^inflate(aRect,-10,-10)
} !!

/*set the new origin point for the object rectangle
from the current mouse position plus offset
Since pt is first initialized to integer,
make sure it only gets integer value*/
Def setNewOriginPt(self, mousePos, offset)
{
  pt.x := asInt(mousePos.x - offset.x);
  pt.y := asInt(mousePos.y - offset.y)
} !!

/*erase the region a little larger than object
rectangle in case it is displayed with a bolded
border*/
Def eraseRect(self,hdc | hBrush)
{
  hBrush := Call CreateSolidBrush(WHITE_COLOR);
  Call FillRect(hdc,inflate(rect(self),5,5),hBrush);
  Call DeleteObject(hBrush)
} !!
```

```

/*check whether the point is contained in the rect*/
Def containedIn(self,point | aBox)
{
  aBox := new(Rect);
  aBox := rect(self);
  if (left(aBox) <= point.x and point.x <= right(aBox)
    and
    top(aBox) <= point.y and point.y <= bottom(aBox))
    ^true
  else
    ^nil
  endif
} !!

```

```

/*computes the difference between the cursor point
  contained in the rectangle and the origin point
  of the object rectangle*/
Def getOffset(self, point | tempPt)
{
  tempPt := new(Point);
  tempPt.x := point.x - pt.x;
  tempPt.y := point.y - pt.y;
  ^tempPt
} !!

```

```

/*returns the default rectanlge dimension for an object*/
Def rect(self | aRect)
{
  aRect := new(Rect);
  init(aRect,pt.x,pt.y,pt.x+140,pt.y+65);
  ^aRect
} !!

```

```

/*draws an object on the window using
  the hdc display context*/
Def display(self,hdc |
  aRect, hBrush, hPen, hOldBrush, hOldPen)
{
  eraseRect(self,hdc); /*first erase it*/
  /*select the color brush for filling
    used with Rectangle (via draw)*/
  hBrush := Call CreateSolidBrush(color);
  /*set bkcolor for shading with DrawText*/
  Call SetBkColor(hdc,color);
  hOldBrush := Call SelectObject(hdc,hBrush);
  aRect := rect(self);
  if thickBorder
    hPen := Call CreatePen(0,5,Call GetTextColor(hdc));
    hOldPen:= Call SelectObject(hdc,hPen);
    draw(aRect,hdc);
    Call SelectObject(hdc,hOldPen);/*restore the dc*/

```

```

    Call DeleteObject(hPen)
else
    draw(aRect,hdc) /*with a reg. border*/
endif;
if nesting /*draw the inner box*/
    draw(nestedRect(self),hdc)
endif;

Call DrawText(hdc,lP(name),-1,aRect,
    DT_CENTER bitOr DT_VCENTER
    bitOr DT_SINGLELINE);
Call SelectObject(hdc,hOldBrush);
Call DeleteObject(hBrush);
freeHandle(name)
} !!

```

APPENDIX L - COLORTAB.CLS FILE

```
/*authorship: Wu, C.T.*/
/*collection of colors available for shading Glad objects*/!!

inherit(OrderedCollection, #ColorTable, nil, 2, 1)!!

now(ColorTableClass)!!

now(ColorTable)!!

/*initialize the table with
  colors available in the system*/
Def set(self | elem, colors)
{
  /*first get the available colors*/
  /*getColorTable() is in ACT dir*/
  colors := getColorTable();
  do(colors, {using(color)
    if color <> 0 and color <> WHITE_COLOR
      /*dont add BLACK or WHITE*/
      elem := new(Array,2);
      elem[USED] := nil;
      elem[COLOR] := color;
      add(self,elem)
    endif}))
} !!

/*returns the next available color for shading*/
Def nextBrushColor(self)
{
  do(self, {using(elem)
    if not(elem[USED])
      elem[USED] := true;
      ^elem[COLOR]
    endif});
  errorBox("E R R O R","No more available color");
  ^WHITE_COLOR
} !!

/*makes the unselected object's color available again*/
Def avail(self, color)
{
  do(self, {using(elem)
    if elem[COLOR] = color
      elem[USED] := nil;
      ^0
    endif}))
} !!
```

APPENDIX M - GLAD RESOURCE SCRIPT FILE

```
/*authorship: Wu, C.T. and Schuett, R.J.*/  
/*FILE GLAD.RC, Contains GLAD menus, dialogs, data,  
and other resource pieces*/
```

```
GladTopMenu MENU  
BEGIN  
  MENUITEM "Create", MAKE_NEWDB  
  MENUITEM "Open", OPEN_DB  
  MENUITEM "Remove", REMOVE_DB  
  MENUITEM "Help", TOPHELP  
  MENUITEM "Quit", QUIT_GLAD  
END
```

```
GladDmlMenu MENU  
BEGIN  
  MENUITEM "Describe", DESCRIBE_OBJ  
  POPUP "ListMembers"  
  BEGIN  
    MENUITEM "All at Once", LIST_MEM  
    MENUITEM "One by One", ONE_MEM  
  END  
  MENUITEM "Expand", CATEGORY_OBJ  
  MENUITEM "ShowConnectn", CONNECT_OBJ  
  POPUP "Change"  
  BEGIN  
    MENUITEM "Add data", ADD_MEM  
    MENUITEM "Delete data", DELETE_MEM  
    MENUITEM "Modify data", MODIFY_MEM  
  END  
  MENUITEM "Query", QUERY  
  MENUITEM "Help", DMHELP  
  MENUITEM "Quit", QUIT_DML  
END
```

```
GladDdlMenu MENU  
BEGIN  
  MENUITEM "Define", DEFINE_OBJ  
  MENUITEM "Modify", MODIFY_OBJ  
  MENUITEM "Help", DDHELP  
  MENUITEM "Quit", QUIT_DDL  
END
```


GladLMMenu MENU

BEGIN

MENUITEM "More", LM_MORE

MENUITEM "Modify", LM_MODIFY

MENUITEM "Help", LMHELP

MENUITEM "Quit", QUIT_LM

END

GladOMMenu MENU

BEGIN

POPUP "Mode"

BEGIN

MENUITEM "Read Only", READ_ONLY

MENUITEM "Edit", EDIT

MENUITEM "Query", QUERY

END

POPUP "Change"

BEGIN

MENUITEM "Add data", ADD_MEM

MENUITEM "Delete data", DELETE_MEM

MENUITEM "Modify data", MODIFY_MEM

END

MENUITEM "Prev", PREV

MENUITEM "Next", NEXT

POPUP "GoTo"

BEGIN

MENUITEM "First", FIRST

MENUITEM "Last", LAST

MENUITEM "I th", ITH

END

MENUITEM "All", LIST_MEM

MENUITEM "Help", HELP_OM

MENUITEM "Quit", QUIT_OM

END

GladCOMenu MENU

BEGIN

MENUITEM "Quit", QUIT_CO

END

ABOUT_GLAD_DIALOG 90,34,122,80

STYLE WS_DLGFRAME | WS_POPUP

BEGIN

CTEXT "GLAD Version 0.01", -1, 23,12,72,11, WS_CHILD

CTEXT "Naval Postgraduate School", -1, 8,25,105,10,WS_CHILD

CTEXT "Dept of Computer Science", -1, 9,37,100,11, WS_CHILD

ICON "glad",-1,26,50,16,16, WS_CHILD

DEFPUSHBUTTON "START", IDOK, 70,58,39,14, WS_CHILD

END

```

OPNDBLIST DIALOG LOADONCALL MOVEABLE DISCARDABLE 70, 23, 166, 102
CAPTION "GLAD Databases"
STYLE WS_DLGFRAME | WS_POPUP
BEGIN
    CONTROL "" DB_LB, "listbox", LBS_NOTIFY | LBS_SORT | LBS_STANDARD |
        WS_BORDER | WS_VSCROLL | WS_CHILD, 5, 16, 110, 82
    CONTROL "OPEN" DEFBUTTON, "button", BS_DEFPUSHBUTTON | WS_TABSTOP |
        WS_CHILD, 125, 17, 33, 13
    CONTROL "ABOUT" ABOUT_DB, "button", BS_PUSHBUTTON | WS_TABSTOP |
        WS_CHILD, 125, 41, 33, 13
    CONTROL "HELP" HELP_LB, "button", BS_PUSHBUTTON | WS_TABSTOP |
        WS_CHILD, 126, 62, 32, 13
    CONTROL "CANCEL" 2, "button", BS_PUSHBUTTON | WS_TABSTOP |
        WS_CHILD, 125, 82, 33, 13
    CONTROL "GLAD Databases" -1, "static", SS_CENTER | WS_CHILD,
        17, 4, 83, 10
END

```

```

RMVDBLIST DIALOG LOADONCALL MOVEABLE DISCARDABLE 70, 23, 166, 102
CAPTION "GLAD Databases"
STYLE WS_DLGFRAME | WS_POPUP
BEGIN
    CONTROL "" DB_LB, "listbox", LBS_NOTIFY | LBS_SORT | LBS_STANDARD |
        WS_BORDER | WS_VSCROLL | WS_CHILD, 5, 16, 115, 82
    CONTROL "REMOVE" DEFBUTTON, "button", BS_DEFPUSHBUTTON | WS_TABSTOP |
        WS_CHILD, 126, 16, 33, 13
    CONTROL "CANCEL" 2, "button", BS_PUSHBUTTON | WS_TABSTOP | WS_CHILD,
        126, 81, 33, 13
    CONTROL "ABOUT" ABOUT_DB, "button", BS_PUSHBUTTON | WS_TABSTOP |
        WS_CHILD, 126, 39, 33, 13
    CONTROL "HELP" HELP_LB, "button", BS_PUSHBUTTON | WS_TABSTOP |
        WS_CHILD, 127, 61, 32, 13
    CONTROL "SELECT the one to be REMOVED" -1, "static", SS_CENTER |
        WS_CHILD, 0, 3, 124, 10
END
!!

```

APPENDIX N - CONSTANTS AND GLOBAL VARIABLES

/*authorship: Wu, C.T. and Schuett, R.J.*/

/*File GLAD.H, Defined Constants and Global variables for GLAD*/

#define SHADE_BOR_WD	20
#define SHADE_BOR_HT	10
#define COLOR	0
#define USED	1
#define NAME	0
#define CLASS	1
#define LENGTH	2
#define USER_DEF	3
#define CATEGORIES	4
#define DEFBUTTON	1
#define MAKE_NEWDB	801
#define OPEN_DB	802
#define REMOVE_DB	803
#define TOPHELP	804
#define DESCRIBE_OBJ	805
#define LIST_MEM	806
#define QUERY	807
#define DMHELP	808
#define DEFINE_OBJ	809
#define DDHELP	810
#define MODIFY_OBJ	811
#define QUIT_GLAD	812
#define QUIT_DDL	813
#define QUIT_DML	814
#define ABOUT_DB	815
#define LM_MORE	816
#define LM_MODIFY	817
#define LMHELP	818
#define QUIT_LM	819
#define ONE_MEM	820
#define READ_ONLY	821
#define EDIT	822
#define ADD_MEM	823
#define DELETE_MEM	824
#define MODIFY_MEM	825
#define FIRST	826
#define LAST	827
#define ITH	828
#define HELP_OM	829
#define QUIT_OM	830
#define PREV	831
#define NEXT	832
#define CONNECT_OBJ	833
#define QUIT_CO	834
#define CATEGORY_OBJ	835

```
#define ABOUT_GLAD          900
#define OPNDBLIST          910
#define RMVDBLIST          911
#define DB_LB              912
#define HELP_LB            913
#define DT_CENTER          1
#define DT_VCENTER         4
#define DT_SINGLELINE      32
#define WHITE_COLOR        0xFFFFFL
#define ES_LEFT            0L
#define ES_MULTILINE       4L
#define MF_DELETE          0x0200
#define MF_INSERT          0x0000
!!
```

APPENDIX O - UNIVERSITY DATABASE FILES

/*authorship of files: C.T. Wu and R.J. Schuett*/

1. UNIVERS.SCH FILE

/*File UNIVERS.SCH, Schema file for University Database*/

DEPT
100@100
N
Name&String&16&S&
Chair&EMPLOYEE&20&U&
Location&String&30&S&
&&

dept.dat
EMPLOYEE
300@100

G
FACULTY
100@100

G
FULL
200@100
N
TenureDate&String&8&S&
&&

ASSOC
400@100
N
ArriveDate&String&8&S&
&&

ASSTPROF
600@100
G
MILITARY
200@100
N
SvcBranch&String&4&S&
DepDate&String&8&S&
&&

CIVILIAN
400@100
N
Status&String&8&S&
&&

@@

Assists&String&15&S&
Courses&String&25&S&
&&

@@
Rank&String&10&S&
Specialty&String&16&S&
&&

STAFF
300@100
G
TYPIST
200@100
N
Type_Speed&Int&8&S&
&&

TECH
400@100
N
Skill&String&15&S&
&&

@@
Rate&Money&8&S&
&&

@@
Name&String&16&S&
Age&Int&8&S&
Pay&Int&8&S&
Address&String&35&S&
WorksFor&DEPT&20&U&
&&

employee.dat
EQUIPMENT
600@100
N
Name&String&15&S&
Ser#&String&10&S&
Belongs&DEPT&20&U&
&&
equip.dat
@@

2. EMPLOYEE.DAT FILE

/*File EMPLOYEE.DAT, Data for Unviersity Employees.*/

1John Smith&25&10,000&123 Maple Ave, Apt 5, Monterey, CA 93904&Astronomy&
2Abe Lincoln&23&23,000&6588 1st Street, Monterey, CA 93940&Biology&
3Joe Doe Jr&14&23,000&8900 Coker Rd, Salinas, CA 93901&Forestry&
4Timothy Collis&44&23,000&9000 Reavis Way, Salis, CA 93901&Forestry&
5John Smith&25&23,000&123 Maple Ave, Apt 5, Monterey, Ca 93904&Astronomy&
6Rob Schuett&36&23,000&9 Gillespie Lane, Monterey, CA 93940&Computer Sci&
7Joe Doe Jr&14&23,000&8900 Coker Rd, Salinas, CA 93901&Forestry&
8Timothy Collis&44&23,000&9000 Reavis Way, Salinas CA 93906&Psychology&
9John Smith&25&23,000&123 Maple Ave, Apt 5, Monterey, Ca 93904&Astronomy&
10Abe Lincoln&23&23,000&6588 1st Street, Monterey, CA 93940&Biology&
11Joe Doe Jr&14&23,000&8900 Coker Rd, Salinas CA 93906&Psychology&
12Timothy Colli&44&23,000&9000 Reavis Way, Salinas CA 90308&Computer Sci&
13John Smith&25&23,000&123 Maple Ave, Apt 5, Monterey, Ca 93904&Astronomy&
14Abe Lincoln&23&23,000&6588 1st Street, Monterey, CA 93940&Biology&
15Joe Doe Jr&14&23,000&8900 Coker Rd, Salinas, Ca 93907&Geology&
16Timothy Colli&44&23,000&9000 Reavis Way, Salinas, Ca 93907&Geology&
17John Smith&25&23,000&123 Maple Ave, Apt 5, Monterey, CA 93904&Astronomy&
18Abe Lincoln&23&23,000&6588 1st Street, Monterey, CA 93940&Biology&
19Joe Doe Jr&14&23,000&8900 Coker Rd, Salinas, Ca 93907&Geology&
20Timothy Collin&44&23,000&9000 Reavis Way, Salinas CA 93906&Psychology&

3. DEPT.DAT FILE

/*File DEPT.DAT, Data for Unversity Departments.*/

Astronomy&Amos Astro&Spanagel 313&
Biology&Ben B. Biolo&Root 334&
Chemistry&Chem Mist Jr.&Spanagel 433&
Chinese Lit&Chu-Chi Chi&Ingersol 130&
Computer Sci&J. J. Hacker&Spanagel 513&
Forestry&Forest DeForest&Wood 3344&
Geology&Geo Geoffrey&Henderson 343&
Mathematics&Real E. Complex&Ingersoll 122&
Psychology&Macho M. Psycho&Root 500&

LIST OF REFERENCES

1. Naval Postgraduate School Technical Report NPS52-87-030, *GLAD: Graphical Language for Database*, by C. T. Wu , July 1987.
2. Naval Postgraduate School Technical Report NPS52-86-014, *A Unified Interface Method for Interacting with a Database*, by C. T. Wu , January 1986.
3. Wartick, Kenneth L., *A Data Definition Language for GLAD*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1986.
4. Duff, Charles, et.al., *Actor Training Course Manual*, Version 1.1, The Whitewater Group, Evanston, Illinois, February 1988.
5. Duff, Charles., et.al., *ACTOR Language Manual*, Version 1.1, The Whitewater Group, Evanston, Illinois, 1987.
6. MacLennan, B. J., *Principles of Programming Languages*, Second Edition, pp. 443-483, Holt, Rinehart and Winston, New York, New York, 1987.
7. Korth, Henry K. and Silberschatz, Abraham, *Database System Concepts*, pp. 45, 171-181, McGraw-Hill Book Company, New York, New York, 1986.
8. Tsurt, Shalom and Zaniolo, Carlo, "An Implementation of GEM - supporting a semantic data model on a relational back-end," *AMC*, v. 84, no. 6 , pp. 286-295, 1984.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Director, Information Systems (OP-945) Office of the Chief of Naval Operations Navy Department Washington, D.C. 20350-2000	1
4.	Superintendent, Naval Postgraduate School Computer Technology Program, Code 37 Monterey, California 93943-5000	1
5.	Professor C. Thomas Wu, Code 52Wq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	5
6.	MAJ Robert J. Schuett Chief, Academic Computing Svc ACD, Office of the Dean West Point, New York 10996-5000	1
7.	MAJ Dana E. Madison Academy of Health Sciences, U.S. Army ATTN: Health Care Administration Division Ft Sam Houston, Texas 78234-6100	1
8.	LTC (RET) Darwin L. Schuett 2018 Linn Street Boone, Iowa 50036	1
9.	Mrs. Ruth E. Faulkner 407 Bonnebrook Street Mundelein, Illinois 60060	1

Thesis
S375115 Schwaneke
c.2 Essentiality weight-
ing models for whole-
sale level inventory
management.

8 NOV 90
5 NOV 91
6 AUG 92

80342
26607
57869

Thesis
S3552 Schuett
c.1 Prototyping visual
database interface by
Object-oriented language.



thesS3552

Prototyping visual database interface by



3 2768 000 84701 6

DUDLEY KNOX LIBRARY